AD-A034 986   MITRE CORP BEDFORD MASS   F/G 15/3

TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS,(U)

DEC 76 M GASSER, S R AMES, L J CHMURA   F19628-76-C-0001

UNCLASSIFIED   MTR-3005   ESD-TR-76-164   NL

1 OF 4

ADA034986

ESD-TR-76-164

(12) B.5

MTR-3005

TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS

DECEMBER 1976

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 522C
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-76-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

Paul A. Karger
PAUL A. KARGER, 1Lt, USAF
Project Engineer

[illegible]
F. WAH LEONG, Major, USAF
Project Officer
Air Force Data Services Center

FOR THE COMMANDER

Frank J. Emma
FRANK J. EMMA, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command & Management Systems

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) ESD-TR-76-164	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS		5. TYPE OF REPORT & PERIOD COVERED
		(14) 6. PERFORMING ORG. REPORT NUMBER MTR-3005
7. AUTHOR(s) (10) M. Gasser, S. R. Ames, Jr., L. J. Chmura		8. CONTRACT OR GRANT NUMBER(s) (15) F19628-76-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522C
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, Bedford, MA 01731		(11) 12. REPORT DATE DECEMBER 1976
		13. NUMBER OF PAGES 331
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 331p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

MULTICS
SECURITY
TESTING

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Modifications and enhancements to the Honeywell Multiplexed Information and Computing Service (Multics) timesharing system have been incorporated to allow Multics to handle multiple levels of classified information. This report contains a description of a mechanism designed to test the security controls at implementation time and at future system updates.

LB

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 235050

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project No. 522C. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

The Air Force Data Services Center (AFDSC) at the Pentagon has recently acquired the Honeywell Multiplexed Information and Computing Service (Multics) to be used in a general timesharing environment. In the past, the system could only operate at a single level of classification because Multics did not support any notion of Department of Defense classifications or clearances. Numerous enhancements to Multics have been provided by Honeywell to allow a multi-level operation in which "non-malicious" users of various clearances can use the system at the same time.

These security enhancements have been incorporated into Multics as part of the standard product for use by any installation having a need for such controls. Although the security features allow open operation at all classification levels, the AFDSC will provide a "benign environment" by administratively limiting access to the system to secret and top secret cleared individuals. Reasonable assurance that the controls function properly will be provided by these test procedures. This report describes the final version of the test procedures.

BACKGROUND

Work is currently in progress to develop a validated kernel-based Multics that can support multi-level operation in an open environment [1]. A security kernel has been implemented in a small operating system for a minicomputer [2], and a validation methodology has been formulated [3]. However, until a validated Multics system becomes available, there is the need for an interim implementation at AFDSC that provides the necessary controls with reasonable assurance that the controls function properly.

A study was undertaken to determine "how secure" Multics was currently, as a guide to the degree of security that may be provided by an enhanced version that incorporates additional security controls [4]. It was determined that Multics as it exists could not be run in an open multi-level environment, but that with certain enhancements and external procedural controls, it could be run in a limited controlled (benign) environment. At the AFDSC the benign environment is provided by limiting access to the system to individuals with at least a secret clearance.

In order to determine in detail what kind of enhancements should be made to Multics, representatives from the Air Force (AFDSC and Electronic Systems Division), Honeywell and MITRE participated in a series of Design Analysis meetings between August and October, 1973. The goal of the Design Analysis was to define a Multics implementation that, through addition of security controls to the existing system, could provide a reasonable degree of security in an unvalidated system, while maintaining as close as possible the existing user interfaces. In fact, to installations that do not choose to use the security controls (e.g., a single level installation), the enhancements should be completely invisible. Also, as far as was feasible, many of the concepts involved in the design of a validated system were to be incorporated into the enhancements, thereby facilitating a smooth transition (for the user) when the validated kernel-based system is implemented.

One of the important concepts incorporated as part of the latter goal is that of the "security perimeter" inside which all program modules are considered "security sensitive". Although there is no security kernel, these security sensitive modules have been identified and design and modification of these components are subject to close scrutiny.

The result of the Design Analysis meetings was a document [5] describing the enhancements that ideally should be incorporated. Due to budget and schedule constraints, however, the actual implementation differs in several minor ways from that described in the document.

DESCRIPTION OF SECURITY ACCESS CONTROLS

Basic Multics Access Controls

The basic Multics access controls allow individual users to control access to information in a discretionary manner through a system of access control lists (ACLs). A user who creates a Multics file, called a segment, can make that segment accessible or inaccessible to other users by specifying in the access control list for that segment the users (or groups of users) to which he wants to grant or restrict access, and by specifying the type of access to be granted to each user. The types of access defined within Multics are: read, write, and execute. Every time a user accesses a segment, his access rights are checked in the ACL for that segment and appropriate privileges are given.

Multics Security Controls

With the incorporation of security controls, further "non-discretionary" access rules are enforced that are not normally controllable by the user. In the military "paper" system each person has a specific security clearance and each document has a classification. A person can only read a document if his clearance is greater than or equal to the classification of the document and if the owner of that document has granted him "need to know" by letting him see it.[1]

The ACL controls in Multics only implement the need to know capability for segments. In order to duplicate the military scheme an additional attribute has been associated with each process and each segment. This attribute of a process is called the "authorization" and the attribute of a segment is called "access class". The terms "authorization" and "access class" are Multics terms synonymous with the military terms "clearance" and "classification". The Multics terms will be used throughout this report.

The clearance of each user is stored in the system and interpreted as a maximum authorization that the user is allowed to use. When the user logs in and verifies his identity through the use of a password, this maximum authorization plus other factors are used to determine the authorization of the process to be created on the user's behalf. After process creation, only the process authorization is used in determining access rights. Processes with the same authorization have exactly the same access privileges (though still subject to ACL controls) even though the respective users may have different maximum authorizations. For example, a user identified by the system as a secret cleared individual may login at the unclassified level and his access privileges are the same as if he himself had no clearance.

Thus, on each access a process makes to a segment, the authorization of the process and the access class of the segment are compared against each other. If both are equal, or if the authorization is greater than the access class, the process may read or execute the segment provided the ACL of the segment allows that process' user to read or execute. The ability to write into a segment is granted only if the process' authorization is equal to the segment's access class,

[1]Classification and clearance as used in this document have two components: a level and a category set. The exact definition of classification and clearance, and the possible relationships between them, will be discussed later (See page 12). In the context of the kernel and validation work, the term "security level" is used to refer to this two-component structure, where "classification" is the first component and category or compartment is the second component.

and if "write" is specified for that user on the ACL of the segment.

The restriction on write access is not a direct military requirement, but is a special case of the *-property, a security-preserving relation that simplifies implementation of security controls [6]. The *-property is actually somewhat broader in that it allows a process to write to segments of a higher access class, but "write up" has been eliminated in the Multics implementation because it is a complication and of no use to users.

Extended Security Controls

Multics contains more than just segments and user processes. The above description of the security controls only satisfies the general military requirements. These controls are ineffective by themselves; security controls must be applied to all areas of Multics.

The Multics file system, which contains all the segments within the system, is a hierarchy as shown in Figure 1. The circles in the figure indicate segments and the rectangles are directories. Directories are special kinds of segments that are not directly accessible to the user, but can only be read or written in an interpretive mode through system procedures. Directories are used to hold the names, locations and other attributes of segments and subdirectories contained within them.

Multics defines three types of access to directories: status, modify, and append, and these are treated in a manner similar to the access modes of segments. A simple extension of segment security controls would allow us to define the types of controls to be placed on directories: "status" can be considered the same as "read", and "modify" and "append" can be treated the same as "write". The application of security controls to directories thus appears to be straightforward. There are, however, several problems involving the various types of control information stored in each directory.

In order to handle many of these problems, the security controls for Multics require that segments within a directory have an access class equal to that of the directory.[2] Directories within a directory must have an access class equal to or greater than that of the parent directory. A directory whose access class is greater than that of its parent is called an upgraded directory. With these restrictions the access classes increase as one goes down in the hierarchy. Information about a segment, such as length, date used, etc., which is

[2]There is the special case of upgraded message segments discussed on page 58.

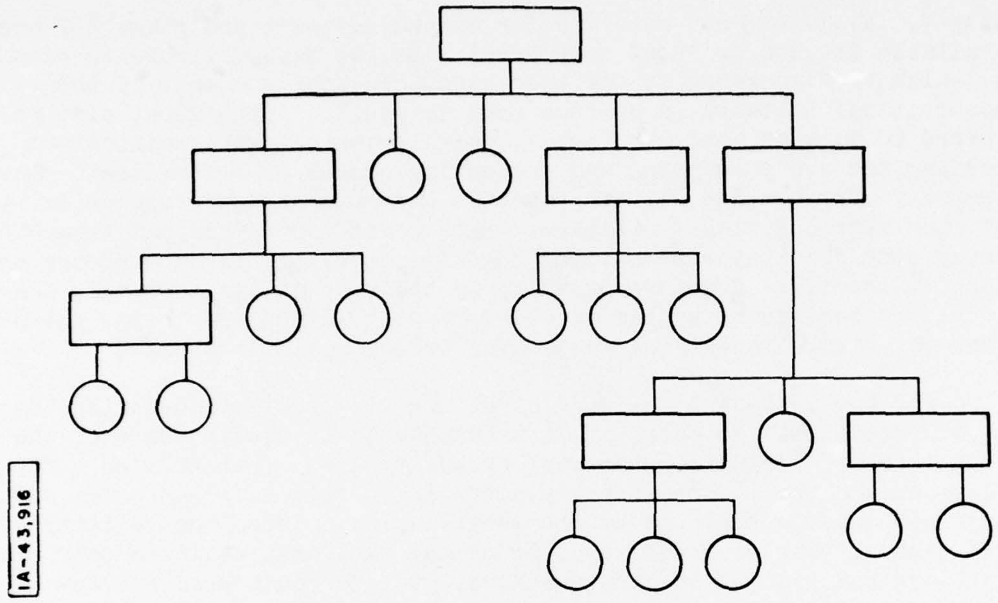


Figure 1. Multics Directory Hierarchy

contained in the parent directory, has the same access class as the segment. Similar information about a directory, however, must be treated in a special way. The special access to directories will be discussed in Section III.

The above contraints on access classes of segments and directories are necessary for the proper enforcement of the security controls. In order to insure that the hierarchy is always consistent with respect to these rules, there is a "security out of service" bit for each segment or directory that can be set by the system if an inconsistency is detected. A user is not allowed to access a directory or segment whose security out of service bit is set. This bit is not used merely to detect bugs in system software -- special processes or privileged users could inadvertently cause an inconsistency to occur. The security out of service bit limits the damage that a user can do in an inconsistent portion of the hierarchy.

Additional Controls

The Multics hardware supports a set of eight ordered protection domains, known as rings, within a process. A process has the greatest privilege while running in ring zero, and the least privilege while in ring 7. Rings 0-3 are reserved for system software and rings 4-7 are available for users. Most user level code, by default, runs in ring 4, which is also known as the user ring. In addition most of the non-critical software in Multics runs in ring 4. Ring zero, also referred to as hardcore, and ring 1, contain the software critical to keeping the system running and protecting access to information. The security controls and all other access controls are the responsibility of ring zero and ring 1 software. User written programs can invoke inner ring primitives by calling specific entry points in hardcore or ring 1 directly. Commands typed in by the user at his terminal, however, are handled by system software running in the user ring, which then may invoke inner ring primitives to accomplish its task.

For the most part the ring structure will be ignored during the test procedures. However, it is necessary to be always aware of the ring in which software under test normally runs. User written software can arbitrarily replace any software in ring 4. Inner ring software can only be replaced by the installation. Thus, the validity of any test of user ring software, or of any test that utilizes user ring software not part of the test package, must be considered in view of the possibility that such software may be purposely or inadvertently bypassed by the user.

DEFINITIONS

The terms "access class" and "authorization" are general terms for Multics that describe how a process' access to an object (e.g. a segment, directory, etc.) is determined. Access class is an attribute of an object and authorization is an attribute of a process. Authorization and access class are actually identically structured -- the different terms are meant only to indicate to what they apply (process or object).

Classification and clearance are very specifically defined in the military. In the computer, the structure of an access class or authorization is represented as follows:

(1) a level number, which is an integer, and
(2) a category set, which is represented in a computer as a string of bits, any combination (or none) of which may be on.

The Multics security controls are designed to operate using any kind of access class structure that may be defined for the installation. However, we will in this report assume that an access class consists of a level and a category component. The test procedures are designed for the most part to handle the above definition of access class, and the AFDSC uses this definition.

Relationships Between Access Classes

Because an access class is more complex than just a level number, the relationships between two access classes must be strictly defined. (The set of access classes is partially ordered.) The possible relationships between two access classes, A and B, are defined below. The notation level(A) and cat(A) refer to the level number of A and the category set of A respectively.

1. A is "less than" B if:

   level(A) < level(B) _and_ cat(A) is a subset of cat(B);
   _or_ level(A) = level(B) _and_ cat(A) is a proper subset of cat(B).

2. A is "equal to" B if:

   level(A) = level(B) _and_ cat(A) = cat(B).

3. A is "greater than" B if:

   level(A) > level(B) _and_ cat(B) is a subset of cat(A);
   _or_ level(A) = level(B) _and_ cat(B) is a proper subset of cat(A).

4. A is "isolated from" B if:

   none of the above.

These definitions are consistent in the normal way in that A "less than" B implies B "greater than" A. Note, however, that not "less than" does not necessarily imply "equal to" or "greater than", since the category sets may be isolated. The specific requirement cited near the bottom of page 9 for read and write privilege assumes the above definitions of these relationships. When one of these four relationships is referred to within this report, it will be written between quotes to avoid confusion with the numeric comparisons between level numbers.

In some places the term "minimum" is used in reference to a group of access classes or authorizations. This term is defined in the usual sense: the minimum of a group of access classes is the "greatest" access class that is "less than" or "equal to" each of the access classes in the group. This minimum can be calculated as the numerical minimum of the levels and the intersection of all the category sets. Note that the minimum of a group of access classes is not necessarily "equal to" any of the members of the group.

Notation

Throughout this report references will be made to specific names of levels and categories that make up an access class. The names for levels are similar to those adopted by the military classification system and were chosen because they are more meaningful than names like "level-1", "level-2", etc. Of course, any names in use at a particular installation whose security controls are to be tested may be substituted. However, if substitutions are made the same relationships between the levels must be maintained throughout in order for the discussion in this report to be consistent.

Each level name has a long and short form that may be used interchangeably. The long and short forms of the level names used within this report are (in increasing order):

unclassified	U
confidential	C
restricted	R
secret	S
top secret	T

These levels need not be exactly one unit apart as long as the ordering is maintained (e.g., there may be additional levels between C and R). The only assumption made in this report is that the lowest level (unclassified or U) is actually equivalent to the lowest level available on the system at the installation. This lowest level is also referred to as "system_low". In addition, "system_high" is used to refer to the highest authorization level with all categories available on the system. It is not important whether the level number of system_high is equal to or greater than top secret.

The names of the categories are arbitrarily defined below:

c1	1
c2	2
c3	3
c4	4
c5	5
c6	6
c7	7

The long forms are the names beginning with "c", and the short forms are just the numbers. There is, of course, no ordering of categories, so any substitutions may be made.

A complete access class or authorization is written as a level name followed by the category names separated by commas, e.g.:

```
        secret,c1,c2,c3
or      S,1,2,3.
```

SECTION II

PHILOSOPHY OF OPERATION

SYSTEM ENVIRONMENT

Purpose and Scope

The main purpose of the test procedures is to check the security control enhancements at initial installation at the AFDSC site and at all new system releases. The procedures are designed to verify that the security controls perform exactly as specified. Thus it is necessary to check both that illegal operations are inhibited and that legal operations are allowed. This latter verification is necessary because a bug that inhibits a legal operation might very well indicate a malfunction that could lead to compromise.

This report describes only the tests of the security controls as enhancements to Multics. Except for one case,[3] the basic Multics controls (access control lists, rings, user authentication, etc.) that have supposedly not been modified from the original Multics design are assumed to work properly. Failure of these latter controls could be an indication of some bug that might lead to a compromise situation, and must therefore be included in a complete test system. However, testing of all of the Multics controls is beyond the scope of this project.

Although the AFDSC will use the enhanced Multics only with secret and top secret cleared users, the tests are designed to check all the controls in a general manner. Thus various levels and categories will be assigned to users and projects. The only installation-specific tests are the I/O tests, where a given system configuration of peripheral devices must be available.

Isolation of Tests

Since all the security controls are implemented in software, it is unnecessary to continuously monitor the system as with a hardware test program -- there is little meaning in running the tests more than once after a system release because software does not deteriorate. There is also no point in trying to detect unauthorized modification to system software by testing, since the assumption is that there are

[3] i.e., the segment ACL controls discussed on page 43.

no malicious users, and a penetrator who can modify system software can easily prevent his modifications from being detected. Of course, there is nothing preventing an installation from running these tests more frequently if it chooses.[4]

Although the entire test system will usually be run as a whole unit, it helps if the various tests can be logically grouped into sections that are isolated from one another so that any one group of tests can be run without running the others. There are several reasons why this grouping is useful.

1. Suspected subversion attempts, bugs or random hardware failures such as disk errors could modify system directories and libraries. If a problem in a particular area is suspected, just those tests for that area can be run.

2. Some of the tests must be done manually and are time consuming. In case of an operator error while running one group of tests, it should not be necessary to restart the entire test sequence, but only to restart that group of tests.

3. Logical grouping helps to localize system bugs. Often a bug in one area will affect other areas. If the results of one group of tests depend on the results of a previous test, and the previous test fails due to a system bug, then further testing is not possible, and other bugs might be undetectable until the previous bug is fixed.

4. Debugging of a new system is greatly simplified if tests can be run that only apply to the area being debugged. (It is not clear whether these test procedures will actually be used in debugging, however.)

5. Logical grouping, of course, simplifies the structure of the tests themselves. This is a very important point due to the fact that test programs cannot usually be completely checked out by just "running" them. To make absolutely sure that a test program will detect a given system failure, one often has to introduce or simulate a system bug. Not all system bugs can be effectively simulated without modification to the system, so source code inspection must be used as a method of debugging. This inspection becomes more reliable as the test programs become less complex and more structured.

---

[4]It must be emphasized that a "trap door" installed in system software by a penetrator may be virtually impossible to detect by any means of testing or inspection. Only validation of software and proper configuration control will assure that trap doors cannot be inserted.

Because each group of tests is independent of the others, functions tested by one group should not be used in another group unless absolutely necessary. For example, the tests for creation of segments are in a different group from those that access the segments themselves. This latter group should not create segments that it intends to access. Rather, a set of already existing segments should be provided. In this way, a bug in the segment creation controls will not manifest itself as a segment access bug. The test environment initialization procedures discussed on page 26 create the segments and other permanent data bases necessary to run the tests.

It may seem more esthetic for a single test program to test everything, creating and deleting all the data bases it might need. But such a program could tend to be very obscure and complex. The extra cost of having segments and directories around permanently is negligible compared to the advantages of simpler test procedures.

It is of course not possible to completely isolate all groups of tests from one another. The mere act of logging in requires creation of many segments and checking of segment access classes. An attempt has been made, however, to minimize the assumptions that one group makes about the performance of features tested in other groups.

Unprivileged Test Programs

All test programs in this series are run at a level of privilege appropriate to that of the area being tested -- mostly at the level of the average user. Some tests can be greatly simplified if they are run at some "system level" with extra privileges.[5] But whenever a test procedure gets more privilege than the area that it is testing, part of the controls are bypassed and proper function of the controls cannot be guaranteed.

Most Primitive Functions

When a user calls a subroutine that eventually invokes some hardcore or inner ring primitive, he often does not know or care whether various controls and tests are made within the primitive function or in higher level software (i.e., user ring software). For example, the user level "delete" command can first check to see if the segment to be deleted exists and whether the user has access (by invoking other hardcore primitives) before calling the hardcore delete subroutine.

[5]For example, the login tests described in Section III could be done automatically if one process had the privilege to login another process.

These checks are a waste of time, however, because the hardcore subroutine must make these checks anyway and returns appropriate status codes. A user can easily write his own delete command that makes no checks before calling hardcore.

When testing security controls it is essential that the controls tested are invoked by direct calls to the most primitive functions available to the user (i.e., calls to hardcore entries). Only in this way is the security of the "system" actually being tested. If one wanted to try to see if it was possible to delete a segment with no access permission he would eventually write a program to call the hardcore delete entry directly rather than give up when the user level delete command refused to do it.

It may often be very convenient for higher level routines to make certain checks before calling the hardcore primitives, but if only the higher level entries are called during testing then there will be some primitive controls that are never exercised.

In most of the tests described within this report, user level system software is bypassed if a security related function is being invoked and if it is possible for the user himself to bypass the system software. Since all commands are user level routines, many commands that are used to test security controls must be rewritten for the test procedures. Some subroutines also have to be rewritten.

System Processes

There is one consequence of testing only the most primitive functions that relates to the Trojan Horse problem [5]. Throughout the course of the Design Analysis it was assumed that procedures within the security perimeter contained no Trojan Horses. Trojan Horses outside the security perimeter in user level software, even if provided as part of the standard system, could not violate security. Thus, no tests of user level software are necessary.

In the case of system processes, which can be considered to lie in the security perimeter, the situation is quite different. A Trojan Horse in what is normally a user level command, when executed by a system process, could result in disaster. This problem is eliminated in a validated system because all system functions that can have an effect on security are included within the kernel and do not depend on any software outside the kernel. For the enhanced Multics, such isolation is not feasible.

An example of a system process is the System Security Administrator (SSA) process. The SSA is given a process with privileges that allow him to change access classes of directories. A Trojan Horse in the SSA command to set the access class of a directory can change the string "secret" to "unclassified" and thereby underclassify a directory. In fact, a Trojan Horse in any of the commands available to the SSA, whether the command is security related or not, can potentially cause considerable damage when executed in a process having special privileges. In order to prevent this occurrence, the SSA is restricted to using only a small subset of specified commands, and these commands, though they may be the same as those generally available to users in an unprivileged mode, must be tested for proper functioning without regard to whether they are the "most primitive". In fact, it is more important to check the highest level commands available to the SSA than any of the more primitive functions.

Auditing

A validated secure system is secure whether auditing takes place or not. The only value of auditing in such a system is in maintaining an accountability of access to controlled data for record keeping.[6] In an unvalidated system, like the enhanced Multics, auditing is also used as an aid to detecting possible security breaches. Auditing in this case can be viewed as a "catch all" in that it has a small chance to detect penetration attempts that may have been allowed due to bugs or omissions in the security controls. In addition, it may be relied upon to detect penetration attempts that may not have been specifically covered in the design of the security controls due to the difficulty of implementation. An example of the latter is the case of message segment overflow, where the "message segment full" condition can be used by a Trojan Horse to transmit one bit of information. During normal operation this condition should occur only infrequently for a given user. By auditing this condition the attempted passage of many bits by this means is detectable. It should be noted, however, that only a very small percentage of all penetrations can ever be detected by auditing -- if a person is clever enough to penetrate the system he can probably cover his tracks by erasing any audit trails that might have been created.

As stated earlier, the main justification for permitting an unvalidated two level system to operate at the AFDSC was that the users of the AFDSC are assumed to be basically trustworthy. Within such an environment, auditing may only detect the less sophisticated (either unintentional[7] or intentional) penetration attempts by AFDSC

---

[6]That is, as far as the security of the system is concerned. There are other reasons for auditing, such as monitoring of performance and recording of vital statistics.

[7]An unintentional penetration on the part of the user is one that occurs accidentally or via a Trojan Horse placed by someone else.

users. The assumption is that a benign user will not go to great efforts to develop a highly sophisticated attack.

It is important to be aware that auditing is only useful if the person or program examining the audit trail is able to sort out meaningful statistics. An example of a penetration attempt for which auditing may be useful, and which the security controls do not cover, is the guessing of another user's password. If every rejected login is audited, it will be obvious from the audit trail, before too long, that someone is making such attempts. The individual examining the audit trail must be able to distinguish this penetration attempt from the normal "accidental" illegal logins that users often make.

Auditing is explicitly tested by performing a specific set of operations that invoke the audit mechanism. An examination of the audit log then indicates whether the operations are properly recorded.

BASIC ASSUMPTIONS

Complete exhaustive checkout of all the security controls in any system is impossible and usually unnecessary. (Of course testing or checkout is theoretically unnecessary in a validated system.) A perfect test system is aware of the details of the system design and only tests each node or decision point in the system once, in a manner similar to checking out hardware logic. With precise implementation details lacking, assumptions must be made regarding this system structure so that a vast number of tests are not required. At the same time one must be wary of making too many assumptions and thus creating an incomplete test system, particularly since the implementation details may change in the future.

The security test system described in this report makes few assumptions about the internal structure of the security controls. The test procedures should be useful for all future system updates, possibly implementing a new or modified design. A future system update must not render the test system incomplete.

The paragraphs below discuss the various types of assumptions one might make about the design of the security controls, and give reasons why these assumptions have or have not been made with regard to the test procedures.

Centralization of Controls

The most obvious assumption is on the centralization of the most basic security controls. The definition of access class used in the

AFDSC system has been given on page 12. In a completely generalized secure system, which is the ultimate goal for Multics, the actual structure of this access class is a system parameter that can vary from one installation to the next. For any particular access class scheme, definitions must be made to allow access classes or authorizations to be compared with one another. The terms "less than", "equal to", etc., were defined on page 13 for the military classification system. Since comparison of access classes is more complex than just a single numeric compare, and since the comparison algorithm may be installation-defined, one would expect there to be a single routine that is called to implement the comparison of access classes. It would also be consistent with Multics philosophy to assume that all instances of access class comparison or testing are handled by the same centralized procedures.

This assumption may not be completely correct, however. Although most of the checks are made by a single subroutine, efficiency, in certain circumstances, may dictate that the checks be explicitly made elsewhere. This may be particularly true because the security checks are often placed at the lowest levels of the operating system. Thus, it is necessary to test that the checks are made properly everywhere the checks are expected to be.

One can still make certain assumptions, however, to simplify the testing. These assumptions can be best described by an example. Using the definitions of the relationships between access classes given on page 13, the PL/I code for determining access to segments would most probably look similar to this:

```
if (authorization.level < access_class.level) or
  not ((authorization.category and access_class.category) =
        access_class.category)
then access_mode = null
else if (authorization.level = access_class.level) and
        (authorization.category = access_class.category)
     then access_mode = full ACL
     else access_mode = ACL minus "write"
```

We assume that it is only necessary to check that each of the operations in the statements above is performed properly. For example, a typographical error, such as ">" appearing in place of "<" should be detectable. It is not necessary to check every possible combination of level and category if it can be assumed that the code used to implement the check is similar to that above. That is, if it works for authorization level 3 and access class level 4, it can be assumed to work in all cases where authorization is less than access class.

Of course, one could always create some strange algorithm that yields right answers for all combinations of levels except one in particular. However, the billions of possible category set and level combinations could never all be tested. We must assume that we have "benign system programmers" who make reasonable attempts to create correct programs. We can only test for accidental omissions or errors.

Since category checks are quite different from level checks (even though both checks may be combined into one PL/I statement) the test procedures are designed to test levels independently of categories with respect to each of the areas tested. The level tests are usually made with null or constant category sets, and the category tests are all made at a single level. In this way, system bugs are much easier to trace down and the test procedures are more straightforward, though perhaps somewhat more time consuming.

Centralization of Functions

There are various "passive" functions associated with the security controls that are repeatedly exercised within the test procedures. A passive function is one that is used to check on the state of something or to return a value but does not otherwise effect any security-related operation. An assumption must be made regarding the believability of values returned by these passive system functions.

For example, there is a subroutine, called hcs_$get_access_class, that returns the access class of any segment. This function must be called several times during the security test procedures. If the access class returned by this function is not believed, a long and arduous sequence of multiple manual logins must be performed in order to infer the access class of a segment. Alternatively, the program wishing to get the access class of a segment can be run in a more privileged state and read the information out of the directory by itself. Both methods are unsatisfactory.

If, on the other hand, one can make the assumption that the same primitive is always invoked whenever the access class of a segment is required, it is only necessary to check that this primitive works once. Furthermore, in this particular example, one can test hcs_$get_access_class merely by requesting the access class of various segments of known access classes that have been previously set up.

Another function belonging to this group is one that converts the character string representation of an authorization to the internal binary representation, and vice versa. This innocuous routine does not even need to run in a privileged state -- any user can replace the system version of this routine with his own. However, if this routine works incorrectly (e.g., returning the string "top_secret" when "secret" was specified) there could be disaster, specifically in the administrative areas and system processes as discussed earlier. We must assume that this same routine is invoked every time the character string-to-binary conversion is required. We only need to test this routine once.

There are other functions performed repeatedly that the test procedures must believe. These will be discussed as necessary in the following section. All such functions whose values are generally believed are explicitly tested.

SUPPORT SOFTWARE

This subsection discusses various support programs used during the security test procedures. These programs are discussed separately from the tests because they are generally called several times in several tests, and thus do not belong to any specific group of tests.

Authorization Tester

The authorization tester is a program that determines the authorization level and category set of the current process (in terms of access rights) and compares this authorization to the authorization that the system thinks the current process has. It prints the current authorization on the terminal.

A special directory is set up during test environment initialization (see Figure 2 on page 31) that contains segments of various access classes. The authorization tester reads segments of successively higher levels and of different category sets. The highest level is calculated and all the categories accessible are or'ed together to determine the process authorization. The authorization tester makes sure that the authorization returned by a call to hcs_$get_authorization (the system primitive that returns the current authorization) is the same as the process authorization that has been determined from access privileges.

The authorization tester does not comprise a test by itself, but is used as a utility during various tests. Its name is "authorization_tester" and it is available as a user callable command that prints the authorization or error message on the terminal.

Access Routines

The authorization tester determines the current authorization by accessing known segments of various access classes. In an analogous manner, there is a program that attempts to access a given segment in different modes (read, execute, write) to determine the type of access available to that segment by the current process. This test provides information as to whether the access class of the segment is "less than", "equal to", "greater than" or "isolated from" the current authorization. There is another program that performs the same function for directories. These programs are in the form of subroutines with the names "try_reference_" and "try_dir_reference_".

SECTION III

DESCRIPTION OF TESTS

This section describes the individual tests that comprise the security test procedures. The tests are grouped by the area of the system that is being tested. The discussion for each group is preceded by a brief description of the design of the Multics security controls to be tested. Some of the tests, designated as scripts, are made manually by an operator, and others are done completely by software. The tests are discussed in this section in sufficient detail for an understanding of the testing process. The complete scripts are contained in Appendix II, and program listings in Appendix IV.

Each test is given an identifier consisting of a series name of three characters followed by a test number (e.g. PAA-5). This identifier can be used to reference the corresponding script in Appendix II. The discussion along with each script indicates the correspondence between program modules and test numbers for those tests performed entirely by software.

TEST ENVIRONMENT

In order to simplify the testing process, a test environment is defined that consists of a set of users, projects, directories, etc. that permanently reside in the system. This environment need be initialized only once per installation. If the tests function properly, data bases defining the environment will not be modified. A system bug, however, could possibly change something and require reinitialization of the environment.

The components of the test environment are users, projects, terminals, I/O devices, and directories. Each of these is discussed below. Appendix I contains detailed instructions for setting up the environment.

Users, Projects and Terminals

These three components of the test environment are initialized by the System Administrator (SA) and the System Security Administrator (SSA). The actual names of the users, projects and terminals need not, of course, be the same as those specified in this section, provided those the same names are consistently used throughout all the tests.

There are four tables maintained by the system that are not normally accessible to the user. These tables contain attributes of users, projects and terminals. A discussion of the security related function of each follows:

1. Person Name Table (PNT)

   This is a per system table that contains an entry for each person (personid) on the system. This table specifies the maximum authorization of each user on the system. This maximum authorization is normally the same as the person's own security clearance.

2. System Administration Table (SAT)

   This is a per system table that contains an entry for each project (projectid) on the system. Each project has a maximum authorization assigned to it that reflects the maximum access class of the work being performed on that project.

3. Project Definition Table (PDT)

   This is a per project table that contains an entry for each user that may login under that project. For each user in a project, this table specifies the maximum authorization that any process for this user may have when running under that project.

4. Channel Definition Table (CDT)

   This is a per system table that contains, among other things, an entry for each channel known to the system. The AFDSC Multics operates only with hardwired terminals (or equivalently, remote terminals over encrypted lines), so that each terminal is known to be attached to a specific physical channel. The channel number uniquely identifies a terminal. The CDT contains the authorization of each terminal, which is determined by the physical access to that terminal: a secret terminal is located in a secret controlled area, etc.

The tables discussed above allow a given user to work on several projects of different authorizations. If the user's security clearance is stored in the PNT as a maximum authorization, addition or reclassification of a project is independent of the security clearance of its users. There is no danger that a project administrator (who can set the authorization of each user in his PDT) will allow a user to run at an authorization greater than that specified in the PNT.

Below are step-by-step instructions for the procedures required to initialize values in these four tables for the users, projects and terminals used in the tests.

1. Create users and projects.

   The System Administrator (SA) creates five dummy projects in the SAT: p1, p2, p3, p4, and p5. There is a single project administrator assigned to all these projects. The SA also inserts seven users in the PNT: u1, u2, u3, u4, u5, u6 and u7.

2. Assign authorizations.

   The SSA assigns the following authorizations to each user and project:

p1: secret	u1: confidential
p2: confidential	u2: top secret
p3: confidential,1,2,3,4,6,7	u3: secret
p4: confidential,4,5,6	u4: confidential,1,3,4,5,6,7
p5: system_high	u5: confidential,1,3,4,5,6,7
	u6: system_high
	u7: system_high

   Five terminals are used. The SSA sets the authorizations of these terminals in the CDT as follows:

   t1: confidential
   t2: top secret
   t3: confidential,1,3,5,6,7
   t4: confidential,1,2,3,4,5,6,7
   t5: system_high

3. Assign authorizations within projects.

   The project administrator logs in under each of the five projects and names all five users under each project. For p1 and p3, the seven users are assigned authorizations within those projects as follows:

   u1: secret
   u2: top secret
   u3: unclassified
   u4: confidential,1,2,4,5,6,7
   u5: confidential,1,2,4,5,6,7
   u6: none
   u7: none

For p2, p4 and p5, no specific authorizations are assigned, thus letting the default take over.

4. Forced generate_password option for u2.

There is a flag that can be set for specific users by the SA that prevents a user from defining his own passwords. Instead, the user is forced to use passwords generated by the system. In order to later test this flag, the SA sets this flag for u2.

As a result of the above five steps, we now have the following authorization values in each of the tables.

PNT	PDT for p1 & p3	SAT	CDT
u1=C	u1=S	p1=S	t1=C
u2=T	u2=T	p2=C	t2=T
u3=S	u3=U	p3=C,1,2,3,4,6,7	t3=C,1,3,5,6,7
u4=C,1,3,4,5,6,7	u4=C,1,2,4,5,6,7	p4=C,4,5,6	t4=C,1,2,3,4,5,6,7
u5=C,1,3,4,5,6,7	u5=C,1,2,4,5,6,7	p5=system_high	t5=system_high
u6=system_high	u6=none		
u7=system_high	u7=none		

Directories and Segments

In Section II it was stated that some of the test procedures require special directories and segments that are set up beforehand. These "subtrees" are part of the static test environment and are created by a series of special commands during the test environment initialization. If the tests function properly, these subtrees should remain intact, since no modifications are made during the test procedures. The test procedures are designed so that aborting during any of the tests will not leave these directories in an unusable state. Even if a security related bug is found during the testing, these directories should not be adversely affected. However, to protect against unforeseen problems, this portion of the test environment should be reinitialized if any system bug is found by the tests.

Five directories are required. Three are referenced for the directory, segment, and System Security Administrator tests, one is used by the authorization tester, and one contains files referenced during the I/O tests. Each of these directories contains subdirectories and segments of various access classes. Creation of these directories manually is a rather tedious process, since the user must repeatedly login (or new_proc) at the different access classes to create segments residing within the upgraded directories. Alternatively, the creator of these directories could perform the whole process at one level of authorization (unclassified), and then exercise system privilege to upgrade the directories and segments to their proper access classes. This latter approach is unattractive since the operation is basically not a privileged one.

Fortunately, Multics provides a mechanism whereby a certain procedure or set of commands, called a start_up.ec, can be executed automatically at the beginning of a process. Given the capability for destroying the current process and creating a new one of a different authorization, most of the procedure can be automated. The exact details are explained in Appendix I. The manner in which the five basic directories are created is not relevant to the discussion below.

1. Directory for authorization_tester.

This directory is shown in Figure 2. In the figure, there is one upgraded directory (and contained segment) for each level and each category within system_high. Thus, in an installation that uses 5 levels and 8 categories there would be 13 directories. Each of the "level" directories is classified at the appropriate level with a null category set. The access class of each of the "category" directories has one category bit set and level number zero. The name of each directory is related to the access class, so that the authorization_tester can reference the proper directory without having to rely on any special primitives that return access classes of objects. If for some reason a directory of some access class within system_high is missing, an error message will be generated when the authorization tester attempts to access the missing directory.

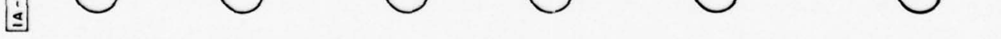


Figure 2. Directory for Authorization Tester

2. Directory for segment security controls tests.

This directory is illustrated in Figure 3. The SEG_TEST directory is at system_low, and each of the subdirectories beneath is at some access class whose category bits and level number bear a specific relation to the authorization at which the segment access tests are to be run. The figure shows specific access classes of the six upgraded directories D1 through D6 as an example, assuming the tests will be run at the authorization secret,c1,c2. The directories DIR and segments SEG are classified the same as their parent. In Appendix I the access classes of the six directories are specified as arguments to the command that creates them. See the specific discussion of the segment security controls test procedures on page 52.

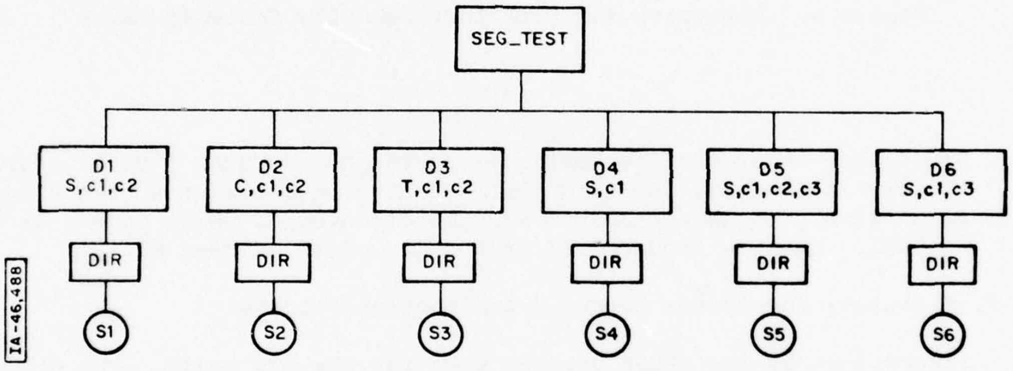


Figure 3. Directory for Segment Security Controls Tests

3. Directory for directory security controls tests.

The directory for the directory tests as illustrated in Figure 4 has almost the same structure as the directory for the segment tests. The only difference is that the segment in each directory is contained directly within the upgraded directory rather than within a subdirectory. See the procedures for directory security controls tests on page 54.

4. Directory for I/O tests.

The directory for the I/O tests, IO_TEST, as illustrated in Figure 5 is at system low and contains segments and directories used in the

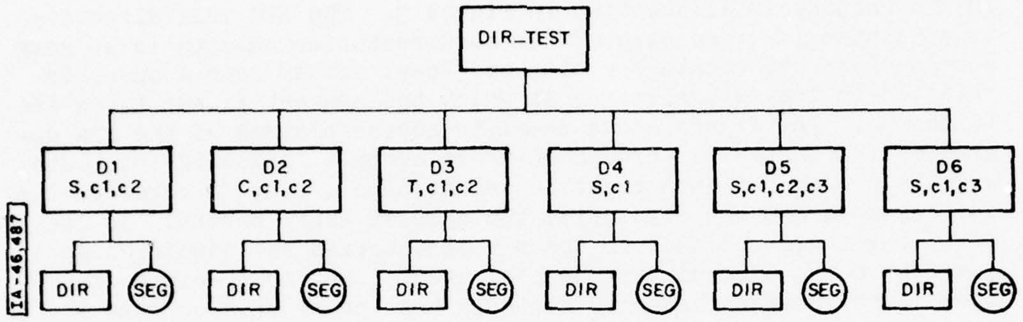


Figure 4. Directory for Directory Security Controls Tests

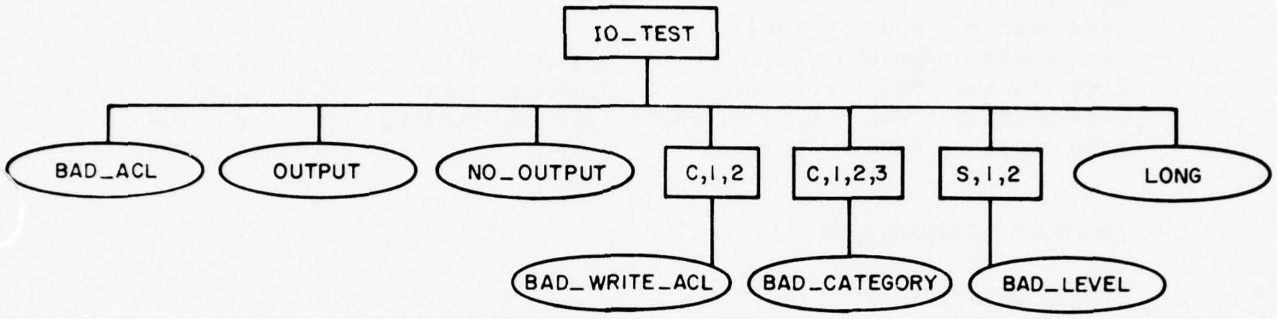
I/O tests. When the I/O tests are performed the user's working directory is IO_TEST. Certain tests require segments of a higher access class. These segments reside in directories whose parent is IO_TEST. See the procedures for the I/O tests on page 61.

5. Directory for System Security Administrator tests.

Figure 6 shows the directory required for the SSA tests. The directory SSA_TEST is at some access class above system_low, and the contained directory and two segments are at the same access class. The segment named MSEG is a message segment (see the discussion of message segments in the tests of communication between processes on page 61). The two segments and the directory are empty.

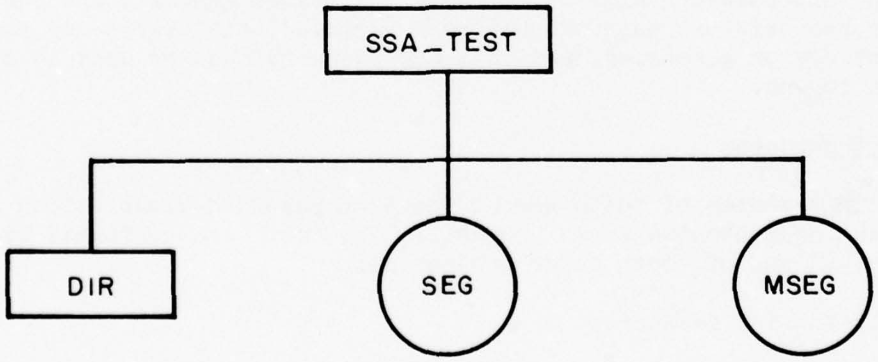
I/O Devices

The initialization of I/O devices is performed by defining device classes having specific values of certain parameters for the required devices. Before each test of a device is made, the device must be logged in and assigned to the proper queue group and device class. The table below lists the values of the relevant parameters required for the I/O device classes to be defined.



1A-46,642

Figure 5. Directory for I/O Tests



1A-46,641

Figure 6. Directory for SSA Tests

device	name	min class	max class	min banner
card reader	crd	(none)	S,c1,c2	(none)
card reader	crd	(none)	U	(none)
card punch	punch	C,c1,c2	S,c1,c2,c3,c4	R,c1,c2,c3
line printer	prt1	C,c1,c2	S,c1,c2,c3,c4	R,c1,c2,c3
line printer	prt2	S,c1,c2,c3,c4	T,c1,c2,c3,c4,c5	T,c1,c2,c3,c4
tape drive	tape	C,c1,c2	C,c1,c2	(none)

PASSWORD DISTRIBUTION

Design Description

The password distribution mechanism in Multics is designed to provide the system with positive proof of the identities of users. Passwords are initially assigned to users by the system administrator, and thereafter a user may change his password at any time. Depending on system parameters, a user's request to change his password may invoke a system procedure to generate a random pronounceable word [7], or the user may be allowed to pick his own new password.

At each login, the user must type his current password on the line following the login line, and the system verifies that the password is correct. If the user wishes to change his password, he indicates this by an option on the login line. The user may elect to change his password himself, or have the system generate one for him. After the original password has been verified, the user's new password is entered or generated, and this new password must be used in subsequent logins.

Test Procedures

This series of tests checks that the password distribution and validation mechanism works properly. The tests are performed by two users, u1 and u2, both under project p1.

PDS-1: Initial password.

The SA has a command that sets a password to a given value for a user. This command is used to assign an initial password to the user when the user is first registered on the system, or at other times such as when the user forgets his password or when it is suspected that the user's password may have been compromised and the user himself cannot be located to change the password. It can also be used to lock out a user (though there are other ways to do that). For this test the SA logs in and changes the password of u1. Then u1 attempts to login with his old password, and this attempt should fail.

PDS-2: Initial password change.

This time the user logs in correctly and changes his password. This test checks that the same initial password as set in PDS-1 is still in force until explicitly changed by the user. Another feature of password control that is checked by this test is the notification to the user when his password was incorrectly used on preceding logins.

PDS-3: Incorrect password entry.

This test checks that the password was actually changed by PDS-2. The user attempts to login, using the initial password, but his attempt should be rejected until he types the new password.

PDS-4: Generate password.

Multics has two password control options that the user may specify when he logs in. The change_password option allows the user to specify a new password for subsequent logins. The generate_password option generates a random password for the user, and allows the user to specify this random password as his new password. Normally, the user has the option of using either of these two methods for selecting a new password. However, there is a parameter can be set that requires all password changes to be made with the generate_password option rather than change_password. Installations such as AFDSC set this parameter because users cannot generally be trusted to pick a password sufficiently difficult for others to guess. For PDS-4, user u1 uses the generate_password option, but chooses to ignore the generated password by entering a different word. Since user u1 does not have the generate_password requirement, this new password is accepted.

PDS-5: Test generated password.

As in PDS-3, the user now logs in again and tries his old password to verify that PDS-4 actually changed his password.

PDS-6: Forced generate password.

This test is very similar to PDS-4, except that this time a different user, u2, uses the generate_password option. Since u2 is required to use the generated password, his attempt to ignore the generated password should fail.

PDS-7: Generate password required.

The final test checks that the change_password option is not permitted for u2. The user attempts to login with the change_password option, but the system should refuse to accept a new password from the user.

PROCESS AUTHORIZATION ASSIGNMENT

This series of tests is designed to check the security controls involved in determining the authorization of a process. Since processes can be created either by a login or at some other time at the request of a user (by the new_proc command), the tests first focus on all the login controls and then concentrate on the new_proc controls. Only the authorization related controls are tested. It is assumed that the normal login controls (password validation, etc.) are working.

Design Description

In the discussion to follow, the term "userid" refers to a specific user on a specific project. The userid is often written as "personid.projectid".

Each possible userid has an authorization that is assigned to it. This userid authorization is the minimum of the values in the SAT, PDT, and PNT for that personid and projectid. Ultimately, we are concerned with the authorization that is assigned to the process created for a userid at login. This process authorization can never be greater than the userid authorization.

The CDT is used to make sure that no terminal transmits data of an access class higher than that of the terminal authorization. The actual authorization assigned to a process for a userid is then further limited by the authorization of the terminal to which the user has logged in.

The user may select, as a login option, to run at any authorization less than or equal to the maximum authorization allowed for his process as determined from the four tables above. When no authorization is specified at login, a user-settable default becomes the authorization of the process. If no default is specified by the user, the lowest authorization -- unclassified with no categories -- is used. The actual authorization of the process is stored in two tables unalterable for the life of the process: the process initialization table and the process data segment. The user-settable default is probably stored in the PNT.

There is one more login test that the system can and does make. Since terminals are in controlled areas, a user whose security authorization is less than that of a specific terminal should not be allowed to use that terminal. If the system detects that the authorization in the PNT for a specific user is less than that of the terminal he is logging in from, a breach of physical security may have occurred and the machine room operator is notified.

Only in this latter case is a message actually printed to the operator. Other illegal logins are simply rejected. All logins, legal or illegal, are audited on the system audit trail.

In the normal Multics environment the user may elect to create a new process for himself and destroy the old one. Creation of a new process is very similar to logging in, and thus, in the AFDSC system, some of the login tests must be retried. When the user wants a new process, he types the "new_proc" command and can optionally select a new authorization at which to run. The system must then validate tne new autnorization in a manner identical to that at login. In the case of an abnormal process termination, in which a new process is automatically created, the authorization of the new process is the same as that of the old.

Every time a new process is created, either at login or new_proc, the authorization of the process is printed on the terminal. This printing cannot and must not be suppressed. Not only is it important that the authorization printed is actually the same as that of the process, but it is very important that this authorization is that selected or expected by the user. If a process of a lower authorization were accidentally created, the user might not notice the printed message (or the terminal might malfunction) and he could possibly think he was running at the higher authorization. He might then input classified information to a process of a lower authorization. Even though this particular malfunction of the security controls does not create a direct compromise situation, it is dangerous and must be checked. In addition, a bug in the controls in this area may indicate a possible bug in some other area that could lead to compromise.

Test Procedures

Since logins cannot be controlled by user software, the testing of logins must be mostly a manual procedure.[8] For these tests, the

[8]MITRE has developed a minicomputer based Remote Terminal Emulator [8], used in testing performance of timesharing systems, that can emulate a large number of terminals. One can prepare scenarios (such as

values in the PNT, PDT, SAT and CDT as set up by the SA and SSA in the test environment will be frequently referred to. Consider the authorizations stored in these four tables. For a given user, project and terminal combination there are many possible relational combinations of the values in these four tables. However, it is only necessary to check that the rule

maximum process authorization = min (PNT, PDT, SAT, CDT)

holds when each of these four values is less than the others. Unfortunately there is no direct (and believable) way to determine what Multics computes as the maximum process authorization. One must first try to login at a higher authorization than this maximum to see if he is rejected; then a login is tried at the maximum process authorization and the login should be accepted.

PAA-1 to PAA-5: Login above maximum.

This first group consists of five logins. Each login checks that the user cannot specify a higher process authorization than the maximum as determined from the PNT, SAT, PDT, and CDT. The table below outlines the five logins and which of the four tables are being tested.

	test for value in	user	project selected	terminal used	auth. selected	maximum process authorization
PAA-1:	PNT	u1	p1	t1	S	C (rejected)
PAA-2:	PDT	u3	p1	t1	C	U (rejected)
PAA-3:	CDT	u2	p1	t1	S	C (rejected)
PAA-4:	CDT	u1	p1	t2	none	** (see below)
PAA-5:	SAT	u2	p1	t2	T	S (rejected)

** The test PAA-4 above tests the additional feature that the operator is notified when the value in the PNT is less than the value in the CDT (breach of physical security). In addition to rejecting the login attempt, the operator's console should print a message.

PAA-6 to PAA-9: Login at maximum.

Now, a legal authorization, which is the maximum authorized, is selected on each of four logins to test each of the four tables. The following table summarizes the tests.

those for logging in) for the emulator that duplicate manual input from the terminal. It may therefore be feasible to use the emulator for the manual procedures described in this section.

	user	project	terminal	authorization selected
PAA-6:	u1	p1	t1	C
PAA-7:	u2	p1	t1	C
PAA-8:	u3	p1	t1	U
PAA-9:	u2	p1	t2	S

The authorization_tester is run after each login to ascertain the authorization of tne new process.

PAA-10 to PAA-12: Login below maximum.

It is necessary to check that the user can specify an authorization that is lower than the maximum process authorization. PAA-10 checks that a confidential process can be created when the maximum is secret. PAA-11 makes sure that the default authorization (when none is specified by the user) is unclassified. In PAA-12, the user logs in and sets his default authorization to confidential. He then logs out and logs in again, this time not specifying an authorization. His default of confidential should be used.

	user	project	terminal	authorization selected	process authorization
PAA-10:	u2	p1	t2	C	C
PAA-11:	u1	p1	t2	none	U
PAA-12:	u2	p1	t2	none	C

The authorization_tester is again run after each of these logins to see that the process authorization is properly set.

PAA-13 and PAA-14: Login at default authorization.

Two more logins check that the PDT default authorizations work properly. For project p2, where the default authorization of each user was not set, the value for p2 in the SAT should become the default.

	user	project	terminal	authorization selected	result
PAA-13:	u2	p2	t2	S	rejected
PAA-14:	u2	p2	t2	C	C process

PAA-15 to PAA-18: new_proc authorization.

The new_proc command can be checked in a single session as summarized in the four tests below. It is assumed that the authorization validation is the same on new_proc as it is at login. Therefore, it is only necessary to test that the controls are invoked and that the selected authorization is properly passed from the previous process.

PAA-15 begins with the user's logging in at secret and forcing an abnormal process termination. The system automatically creates a new process in such a case, and the authorization of this new process (as verified by the authorization_tester) should also be secret.

In PAA-16 the user, still running the secret process, manually creates a new process and specifies an authorization of confidential. This tests that the user is able to downgrade his new process properly.

PAA-17 is similar to PAA-16 except that this time the user attempts to upgrade his process from confidential to secret.

The last new_proc test, PAA-18, checks that the user is not allowed to create a process above his maximum. In this test the user attempts to new_proc to top secret, and his attempts should be rejected. The rejection in this last test is due to the user's exceeding the authorization specified for p2 in the SAT. It is assumed that the CDT, PDT, and PNT are also limiting factors as they were for login. No further checks are made with respect to these.

user-terminal-project	current authorization	action performed	new process authorization
PAA-15: u2 t2 p1	S	abnormal term.	S
PAA-16: u2 t2 p1	S	new_proc to C	C
PAA-17: u2 t2 p1	C	new_proc to S	S
PAA-18: u2 t2 p1	S	new_proc to T	rejected

In PAA-18 a special new_proc command is used, instead of the standard system new_proc. This special new_proc operates the same as the system's new_proc, except that it exercises the system primitives in ring zero by making no checks for authorization errors in the user ring as might be made by the system new_proc command.

PAA-19 to PAA-29: Category tests at login.

Up to this point, the security controls were tested only with respect to levels -- null category sets were used. It is necessary to test that the category sets are handled properly in each of the tests PAA-1 through PAA-18.

Since categories are only partially ordered, several possible relationships can exist between two category sets C1 and C2:[9]

C1 "greater than" C2
C1 "equal to" C2
C1 "less than" C2

or, if none of the above:

C1 & C2 = null (disjoint)
C1 & C2 = not null (isolated but not disjoint)

The maximum process authorization defined near the top of page 38 should be properly calculated for each of the above cases. Since the rule specifies calculation of a minimum, and the minimum is calculated as the intersection of the category sets, it is necessary to determine that each of the category sets (in the PNT, SAT etc.) are included in this calculation. All the following tests are made at a single authorization, confidential, because it has already been verified that levels are handled properly.

User u4's maximum process authorization contains only the categories 1,6,7 when he logs in from terminal t3 on project p3. If any one of the four tables is left out in the calculation of this maximum, the process authorization will contain extra categories. As in the previous tests, the user must login and chose a "greater", "equal" and "lower" category set to determine which categories are actually set. Unfortunately, if more than one of the categories chosen by the user on the login line is not within the calculated minimum authorization, the login will be rejected and there will be no way to determine which of the categories were illegal.[10] Therefore, several logins must be tried.

The table below summarizes the 11 logins that are to be attempted. PAA-19 to PAA-22 are rejected because one of the categories specified is not included in one of the four tables. The first four tests thus check that each of the tables are included in the category verification. PAA-23 determines that three of the categories that appear in all the tables are indeed included. PAA-24 determines that a user can select a category set that is a subset of the maximum authorized. PAA-25 checks the default case of unclassified, no categories. PAA-26 checks the user settable default. For this test, the user logs in and picks a default authorization of C,6,7. On the next login, this default should be used. PAA-27 and PAA-28

[9]See definitions on page 13.

[10]The answering service, which reads the user's login line, does not notify the user of his maximum allowable authorization. Even if it did, however, such a message could not be trusted to be correct.

test the PDT defaults for p4, and PAA-29 checks that the operator is notified when a category in the CDT is not in the PNT.

	user project terminal	login cat.	result
PAA-19:	u4 p3 t3	1,2,6,7	rejected
PAA-20:	u4 p3 t3	1,3,6,7	rejected
PAA-21:	u4 p2 t2	1,4,6,7	rejected
PAA-22:	u4 p3 t3	1,5,6,7	rejected
PAA-23:	u4 p3 t3	1,6,7	accepted
PAA-24:	u4 p3 t3	1,6	accepted
PAA-25:	u4 p3 t3	none	process authorization U, no category
PAA-26:	u5 p3 t3	default	process authorization C,6,7
PAA-27:	u4 p4 t3	4,5,6,7	rejected
PAA-28:	u4 p4 t3	4,5,6	accepted
PAA-29:	u4 p3 t4	none	rejected, notify operator

PAA-30 to PAA-33: Category tests at new_proc.

The final series of tests checks the new_proc options identically to that in PAA-15 to PAA-18. As in PAA-15 to PAA-18, it is assumed that the PNT, PDT and CDT values of the maximum category set cannot be exceeded. The table below summarizes the tests made.

	user-proj-term	auth.	action	new authorization
PAA-30:	u4 p3 t3	1,6,7	abnormal term.	1,6,7
PAA-31:	u4 p3 t3	1,6,7	new_proc to 1,6	1,6
PAA-32:	u4 p3 t3	1,6	new_proc to 1,7	1,7
PAA-33:	u4 p3 t3	1,7	new_proc to 1,4	rejected

PAA-34: Default too large.

The default authorization as set by the user in PAA-26 should still be in force. Since the default should apply to the user no matter which project or terminal he uses, it is possible that this default may be greater than that allowed for some project. For this test, user u5, who currently has a default of C,6,7, tries to login under project p4, which has an isolated category set. This login should be rejected.

ACCESS TO SEGMENTS

Although there are other types of objects within Multics to which access is controlled, segments can be considered most basic because access to them is monitored directly by hardware. Because these direct segment access controls are so fundamental (many other types of access control depend on them), it was judged necessary to test the existing Multics "need to know" controls for segments as well as the new security controls. The tests of the two types of controls are entirely distinct and will therefore be discussed separately within this subsection under the headings "ACL Controls" and "Security Controls".

Design Description - ACL Controls

As briefly mentioned earlier, each segment in the system has an Access Control List (ACL) that specifies the types of access any user of the system has to that segment. When a segment is first referenced by a process, the ACL of the segment is searched, and if at least one of the three access control bits (read, execute or write) is on for the current user, the segment may be "initiated". During initiation a segment descriptor word (SDW) is created containing a pointer to the segment and the three access control bits from the ACL that apply to the current user. This SDW is referenced by the hardware on every machine instruction that accesses the segment. If an instruction is executed that attempts a type of access to that segment not allowed by the access control bits in the SDW, a fault occurs and the operation is inhibited.

The proper functioning of hardware with respect to the SDW access control bits is tested by a hardware "subverter," discussed in another document [9]. Though some of the hardware tests are effectively duplicated by the procedures discussed here, the purpose of the ACL control tests is to verify that the supporting software in hardcore that maintains ACLs (setting, listing, searching, etc.) works properly. Also, since a great deal of interpretive ACL searching and validation is performed by software before the SDW is created during initiation, the proper functioning of this software must be verified. The following paragraphs discuss the Multics ACL mechanism as it appears to the average user. This information is extracted from the Multics Programmers' Manual [10], and gives an idea of the types of operations software must perform in the maintenance of ACLs. Hardcore is completely responsible for the maintenance of ACLs as described here. The only role played by user level software is in providing a command level interface to hardcore.

The ACL on a segment created by the user is a linear list of "entries". Each entry is composed of a "group identifier" and access mode indicators. The group identifier delineates a set of Multics processes and is made up of three components as represented below:

user.project.tag

The user and project are character strings, and the tag is a single character indicating a process type. Any of these three components can also be the single character "*". The access mode that corresponds to a group identifier may be any combination of read, execute and write (r, e, w) or null (n). As an example, the ACL of a segment may appear as follows:

Drone_1.Blithe.a	rew
Drone_2.Kith.*	re
.Kith.	rew
.SysDaemon.	n
..*	r

The ordering of entries with respect to the "*" components is important. When an ACL is sorted, components consisting of "*" are considered to follow corresponding components not consisting of "*", where the sorting is by the three components, left to right.

Every process in Multics has a permanently assigned, non-forgeable access identifier. This access identifier is composed of the user's name, project, and process type as is a group identifier, except that "*" is not used. For example, the user Drone_2, logging in from a terminal under the project Kith, is given a process with the access identifier Drone_2.Kith.a, where the tag "a" signifies an interactive process.

If the user Drone_2 now wants to access a segment having the ACL shown above, a search of the ACL is made for a match with the process identifier. In this search, components of the group identifiers in the ACL consisting of "*" are considered to match any corresponding component of the process identifier. The access mode of the first ACL entry that matches is the process' access to the segment. In the example, the first match was with the second entry "Drone_2.Kith.*", so the access is "re". Note that the third entry would have applied to this process if the second entry was not there. In this particular example all users under the project "Kith" have "rew" access except the user "Drone_2", who has only "re" access. If the mode for a process is "null", or if there is no match in the ACL, no access to the segment is permitted and the segment may not be initiated. Otherwise, an SDW is created and the mode bits from the ACL are copied into it. From this point on the hardware takes over in controlling access to the segment on each instruction.

<u>Test Procedures - ACL Controls</u>

In order to manipulate ACLs of segments, the user need only have modify permission on the containing directory. The actual contents of the ACL entries are left entirely up to the discretion of the user.

The user normally manipulates ACLs by calling commands that provide interfaces to the hardcore primitives that perform the function desired. Though it is important that the user level commands work properly, only tests of the actual hardcore primitives are made. If a user does not trust the system provided ACL commands, he can always bypass them and call these primitives himself.

There are five primitive functions to create, add to, delete from, list, and replace segment ACLs:

hcs_$append_branch	create
hcs_$add_acl_entries	add
hcs_$delete_acl_entries	delete
hcs_$list_acl	list
hcs_$replace_acl	delete/create

A series of automated tests can check that all five functions perform as expected. These tests are broken up into five groups. The first four groups check the mutual consistency of hcs_$list_acl with the other four primitives, and the last group checks that the ACL as prepared is properly copied into the SDW and enforced. There are also checks to ensure that the ACL cannot be made to contain "garbage" that might confuse the system into misinterpreting the ACL. For these tests the user u1 logs in under project p1 from terminal t1. Since security controls are being ignored for these tests, the entire test sequence can be assumed to take place at system_low or some other single authorization level. Note that if any error is detected in these tests, then the <u>entire</u> test of the ACL controls is terminated with an appropriate error message. This is because the impact of a detected error in the ACL controls is difficult to determine. In the paragraphs below, a brief description of the hcs_ entry point being tested precedes the discussion of each group.

SAC-1: Consistency of hcs_$append_branch with hcs_$list_acl.

The primitive hcs_$append_branch is used to create a segment and to initialize the ACL of the segment to a certain "initial ACL" plus the group identifier for the current user and project with a specific access mode. The initial ACL is a special ACL obtained from a list stored in the containing directory. The initial ACL itself is maintained with a set of primitives similar to the segment ACL primitives, but they are not tested in this series. The default initial ACL for segments is empty. However, hcs_$append_branch also automatically gives "rw" access to all SysDaemons. Thus, assuming the user u1 creates a segment using hcs_$append_branch, specifying his access mode as r, the resulting ACL should appear as follows:

u1.p1.*	r
.SysDaemon.	rw

For this test a segment is created with the above ACL and then hcs_$list_acl is called to check this ACL.

SAC-2 to SAC-7: Consistency of hcs_$add_acl_entries and hcs_$list_acl.

The primitive hcs_$add_acl_entries adds or changes entries in an already existing ACL. For this group, a segment is created as in SAC-1 with the following ACL:

u1.p1.*	rw
.SysDaemon.	rw

Six attempts then are made to add entries to this ACL. These attempts check that hcs_$add_acl_entries does nothing when supplied badly formed ACL entries, but correctly changes or inserts when supplied well formed ACL entries. After each attempt, a check is made to verify that hcs_$list_acl yields the ACL expected. Part of the test is to verify that the additional ACL entries are inserted into the proper place in the ACL, since the order is very important. The following table summarizes these six tests:

	Additions		Result	Resultant ACL	
SAC-2:	u1.p2.*	r	No entries	u1.p1.*	rw
	a.b.c.d	rew	added	*.SysDaemon.*	rw
SAC-3:	u1.p2.*	r	Entry added	u1.p1.*	rw
				u1.p2.*	r
				.SysDaemon.	rw
SAC-4:	u1.p2.*	re	Entry	u1.p1.*	rw
			changed	u1.p2.*	re
				.SysDaemon.	rw
SAC-5:	u2.p2.*	re	Entry added	u1.p1.*	rw
				u1.p2.*	re
				u2.p2.*	re
				.SysDaemon.	rw
SAC-6:	u2.p2.b	rew	Entry added	u2.p2.b	rew
				u1.p1.*	rw
				u1.p2.*	re
				u2.p2.*	re
				.SysDaemon.	rw

	Additions		Result	Resultant ACL	
SAC-7:	*.p1.*	r	Entries	u2.p2.b	rew
	u2.*.*	r	added and	u1.p1.*	rew
	u1.p1.*	rew	changed	u1.p2.*	re
	..*	e		u2.p2.*	re
				u2.*.*	r
				.SysDaemon.	rw
				.p1.	r
				..*	e

SAC-2 should add no entries because there is an error in one of the entries to be added (four components a.b.c.d instead of three). SAC-3 checks that one new entry that has the same user and tag, but a different project, as another entry is added in the proper place. SAC-4 checks that "adding" an entry for a group identifier already on the ACL results in replacement of that entry with the new access mode. SAC-5 adds a new entry which is the same as another but differs in user name. SAC-6 adds a new entry that differs only in the tag from some other entry. SAC-7 adds a list of four entries, purposely out of order, to check that each is properly inserted into the ACL.

SAC-8 and SAC-9: Consistency of hcs_$delete_acl_entries and hcs_$list_acl.

The function hcs_$delete_acl_entries deletes specific entries from an ACL. A segment is first created with the following ACL:

u1.p1.a	rew
u2.p2.a	rew
u2.p3.a	re
u1.p1.*	rw
.SysDaemon.	rw
.p2.	r

Two attempts are then made to delete entries from this ACL. These attempts check that hcs_$delete_acl_entries does nothing when supplied badly formed ACL entries, but correctly ignores or deletes when supplied well formed ACL entries. After each attempt, a check is made to verify that hcs_$list_acl yields the ACL expected. The following table summarizes this group. Note that in SAC-8, the illegal entry has fewer than three components, instead of more than three as in SAC-2.

	Deletions	Result	Resultant ACL	
SAC-8:	u1.p1.*	No entries	u1.p1.a	rew
	u2.p2.a	deleted	u2.p2.a	rew
	u3.p4.*		u2.p3.a	re
	.SysDaemon.		u1.p1.*	rw
	a.b		*.SysDaemon.*	rw
			.p2.	r
SAC-9:	u1.p1.*	Legitimate	u1.p1.a	rew
	u2.p2.a	entries	u2.p3.a	re
	u3.p4.*	deleted	*.p2.*	r
	.SysDaemon.			

SAC-10 to SAC-12: Consistency of hcs_$replace_acl and hcs_$list_acl.

The primitive hcs_$replace_acl replaces an entire ACL. It is equivalent to using hcs_$delete_acl_entries for every entry and then calling hcs_$add_acl_entries with a user-supplied list. A segment is created with the following ACL:

u1.p1.*	rw
.SysDaemon.	rw

Three attempts are made to replace this ACL with another ACL. These attempts check that hcs_$replace_acl does nothing when supplied a replacement ACL with badly formed ACL entries, but correctly constructs a new ACL when supplied a replacement ACL with all well formed entries. After each attempt, a check is made to verify that hcs_$list_acl yields the ACL expected. The following table summarizes this group.

	Replacement		Result	Resultant ACL	
SAC-10:	u1.p1.a	rew	ACL replaced	u1.p1.a	rew
	..*	r		u2.p2.*	rw
	.SysDaemon.	rw		*.SysDaemon.*	rw
	u2.p2.*	rw		*.*.*	r
SAC-11:	u3.*.*	r	No replace	Unchanged	
	a.b.c.d	rew			
SAC-12:	(empty ACL)		ACL replaced	(empty ACL)	

SAC-13 to SAC-25: Control of access.

Having completed successfully the first four groups, mutual consistency of the function hcs_$list_acl with the other primitive functions is assured. The final group, consisting of SAC-13 through SAC-25, checks that an ACL of a segment as set by the four primitives does control correctly the access of a process to that segment. To do this, a different user u2 under a project p2, logs in at a second terminal t2. Let P1 and P2 indicate the processes for u1 and u2 respectively. User u1 first creates a segment with a specific ACL as shown in the table below under SAC-13. Using the primitives hcs_$add_acl_entries, hcs_$delete_acl_entries, and hcs_$replace_acl, a series of changes are made to this ACL. After each change, a check is made to verify that the access that process P2 is given to the segment is consistent with the current ACL. The check consists of using the try_reference_ subroutine described on page 24 to access the segment. This particular ACL and series of changes were chosen also to ensure that, when the ACL of a segment is examined in order to determine the access allowed a particular process, the process is associated with the correct ACL entry. These thirteen tests are summarized in the table below. Under the heading "Entry" the hcs_ entries used to change the ACL are named. Under ACL, either the entire new ACL is shown (when "result:" is indicated), or the specific entries added or deleted are named.

	Entry	ACL		Access of P2
SAC-13:	(append & add)			
	result:	u2.p2.x	rew	null
		u2.p3.a	rew	
		u3.p2.a	rew	
		u2.p2.a	null	
		u1.p1.*	rew	
		u2.p2.*	rew	
		u2.*.a	rew	
		u2.*.*	rew	
		*.p2.a	rew	
		.SysDaemon.	rw	
		.p2.	rew	
		..a	rew	
		..*	rew	
SAC-14:	add:	u2.p2.a	r	r
SAC-15:	add:	u2.p2.a	r	re
SAC-16:	add:	u2.p2.a	rw	rw
SAC-17:	add:	u2.p2.a	rew	rew

	Entry	ACL		Access of P2
SAC-18:	(delete & add)			
	result:	u2.p2.x	rew	r
		u2.p3.a	rew	
		u3.p2.a	rew	
		u1.p1.*	rew	
		u2.p2.*	r	
		u2.*.a	rew	
		u2.*.*	rew	
		*.p2.a	rew	
		.SysDaemon.	rw	
		.p2.	rew	
		..a	rew	
		..*	rew	
SAC-19:	delete:	u2.p2.*	r	r
	add:	u2.*.a	r	
SAC-20:	delete:	u2.*.a	r	r
	add:	u2.*.*	r	
SAC-21:	delete:	u2.*.*	r	r
	add:	*.p2.a	r	
SAC-22:	delete:	*.p2.a	r	r
	add:	*.p2.*	r	
SAC-23:	delete:	*.p2.*	r	r
	add:	*.*.a	r	
SAC-24:	(delete & add)			
	result:	u2.p2.x	rew	r
		u2.p3.a	rew	
		u3.p2.a	rew	
		u1.p1.*	rew	
		.SysDaemon.	rw	
		..*	r	
SAC-25:	replace:	u2.p2.x	rew	null
		u2.p3.a	rew	
		u3.p2.a	rew	
		u1.p1.a	rew	

In SAC-13, process P2 should be associated with the fourth entry in the list, and therefore should be given null access. The first three entries each match P2's access identifier of u2.p2.a in exactly one of the three components, and the fifth does not matcn p2 in any component. The other entries all match P2 in all three components, utilizing different positions of the "*" identifier (except the one for *.SysDaemon.*), but they should be ignored because they follow the first match with u2.p2.a. SAC-14 through SAC-17 check that the different combinations of modes are enforced properly. Only the four generally useful combinations are checked, rather than all possible combinations. In SAC-18 through SAC-24, the first matching entry from each previous test is deleted from the ACL, and the next entry's mode is changed to "r". These verify that components of "*" are properly matched. Finally SAC-25 checks that there is null access when there is no match.

Design Description - Security Controls

With the application of security controls to segment references, the access mode as determined by the ACL on the segment may be further restricted. The security controls can be thought of as being applied to the three mode bits (r, e, w) just before they are put into the SDW. As expressed in the representation of PL/I code at the middle of page 22, these controls state that

1) if the authorization of the process is "equal to" the access class of the segment, leave the mode unchanged;

2) if the authorization is "greater than" the access class of the segment, subtract "w" access;

3) in all other cases, the access to the segment is null, and the segment is not initiated.

There are further complications with regard to segment access involving the ring structure as discussed in Section I. The ring structure imposes additional controls on access to segments, and is enforced by hardware utilizing additional fields in the SDW. Most of the hardware supported ring structure is tested by the hardware subverter discussed earlier. There are also commands in support of the ring structure as for ACLs (for example, each segment has a set of ring brackets that can be set by using certain commands) but the decision was made not to test these because the only way a user can set the ring brackets of a segment below his own validation level is to have access to a special "gate" segment that is protected by the ACL controls already tested. Also, bugs in these commands are unlikely since the interface is quite simple. It is possible for the user to create his own subsystems using rings as a protection mechanism, but since there is no interaction between the ring structure and the security and ACL controls, bugs in the user's subsystem can only involve data to which he already has access. Such a subsystem would have to be thoroughly tested before it could be relied upon to protect data via the ring mechanism.

Test Procedures - Security Controls

Many of the segment access checks have already been performed by the authorization tester during the process authorization assignment tests. The segment test procedures, however, do not assume the authorization tester has been run, and thus are entirely independent of any other group of tests. These tests can all be automated with no manual intervention required.

The SEG_TEST directory set up during test environment initialization described on page 31 (Figure 3) is referenced in these tests. All tests are performed at a single login session, authorization secret,c1,c2. Immediately after login, the test program is called to perform all the tests as outlined in the table below. In this table, S indicates the access class of the segment and P is the authorization of the process (secret,c1,c2).

	Test	Attempted access	Segment	Result
SSC-1:	S=P	write	S1	access allowed
SSC-2:	S=P	read	S1	access allowed
SSC-3:	S=P	execute	S1	access allowed
SSC-4:	S<P	read	S2	access allowed
SSC-5:	S<P	execute	S2	access allowed
SSC-6:	S<P	write	S2	access denied
SSC-7:	S>P	initiate	S3	access denied
SSC-8:	S⊂P	read	S4	access allowed
SSC-9:	S⊂P	execute	S4	access allowed
SSC-10:	S⊂P	write	S4	access denied
SSC-11:	P⊂S	initiate	S5	access denied
SSC-12:	S≠P	initiate	S6	access denied

In the above table, the symbols "<" and ">" refer to the comparison of level numbers, category sets being equal. The symbols "⊂" and "≠" mean "subset" and "isolated" in reference to category sets, with level numbers equal.

The tests listed are self explanatory. References to the segments are made using the subroutine try_reference_. Tests SSC-1 through SSC-7 check the relationships between level numbers, and SSC-8 through SSC-12 check the relationships between category sets.

ACCESS TO DIRECTORIES

For directories Multics includes ACL controls, ring controls and security controls that operate in a manner similar to that for segments. However, all of these controls are enforced by software rather than by hardware. In order to reasonably limit the scope of testing, only the security controls are tested.

Design Description

Though similar to the segment controls, the directory access controls are in reality much more complex. This complexity stems from the fact that directories are never directly referenced by the user, but are referenced through hardcore in an interpretive manner. Instead of setting bits in an SDW the first time the segment is referenced at initiation time, the hardcore supervisor must verify that the user has access on every call to every hardcore primitive that accesses directories. An additional complication is that, while the access class of a segment must be the same as that of its parent directory (and therefore information about the segment (name, length, etc.) stored in the directory is of the same access class as the segment contents), the access class of a directory may be greater than that of its parent.

Every time a segment is initiated its parent directory must be accessed. Without security controls, it is not necessary for a user to have any access to the parent directory (as specified in the ACL of that directory) in order to access the segment, as long as he has some access to the segment. Even though there is no access to the directory, however, there is implicit access to various items in the branch for the segment. (The term "branch" is used to refer to the attributes of the segment or directory stored in the parent directory.) The bit count of a segment, for example, may be obtained from the parent directory with any access to the segment. There is also an implicit "write" access to items such as the date time used (dtu) which are modified by the system when the segment is accessed. Thus, it is possible, even with no access to a directory, to examine and even modify items stored in that directory.

The *-property requires that it must not be possible for a process of a higher authorization to write data that can be read by a process of a lower authorization. In a typical case where a secret process accesses an unclassified segment (contained in an unclassified directory) the secret process normally should have no "write" access to the segment or the directory. Clearly, it would be a violation if attributes such as the dtu of the segment were modified and then readable by unclassified processes. With the incorporation of security controls it has become necessary to restrict implicit modification of items in a directory having an access class below the authorization of the process.

Test Procedures

The three directory access modes "status" (s), "modify" (m) and "append" (a) are, for the purposes of security, considered equivalent to the segment access modes as follows:

$$s = r, e$$
$$m, a = w.$$

The first group of directory access tests is exactly the same as the test of the segment security controls, except that there is no explicit test of initiate for a directory, and "a" access is not tested separately. The subroutine try_dir_reference_ is used to make the directory accesses. This subroutine, when given the name of a directory to access, tries to use every hcs_ (hardcore) primitive (documented in the MPM [10]) to access that directory. In each case, both "s" and "m" access modes are checked. This first group of tests is outlined in the table below. Refer to Figure 4 on page 32 for an illustration of the directories referenced in these tests.

	Test	Directory	Mode allowed
DSC-1:	D=P	D1	sm
DSC-2:	D<P	D2	s
DSC-3:	D>P	D3	null
DSC-4:	D∈P	D4	s
DSC-5:	P∈D	D5	null
DSC-6:	D≠P	D6	null

In this table the symbol D indicates the access class of the directory being referenced. The meanings of the other symbols are the same as those in the table for segments on page 52.

The above tests only check access to entries that already exist within directories of various classifications. Several more tests are required to check an additional primitive hcs_$create_branch_ that may be used to create directories or segments of any access class. It must be verified that hcs_$create_branch_ cannot be used to create illegal hierarchy configurations, and that it also cannot be used to pass information. Each of the tests DSC-7 to DSC-17, summarized in the table below, attempts to create a directory or segment of a certain access class within another directory. For these tests, it is again assumed that the process authorization is secret,c1,c2.

	parent directory		entry created			reason for
	name	access class	name	access class	quota	failure
DSC-7:	D2	C,c1,c2	dir	S,c1,c2	1	no "m" to D2
DSC-8:	D2	C,c1,c2	dir	S,c1	1	no "m" to D2
DSC-9:	D4	C,c1	dir	C,c1	1	no "m" to D4
DSC-10:	D3	T,c1,c2	dir	S,c1,c2	1	no "s" or "m"
DSC-11:	D5	C,c1,c2,c3	dir	S,c1,c2	1	no "s" or "m"
DSC-12:	D1	S,c1,c2	dir	S,c1	1	downgraded dir
DSC-13:	D1	S,c1,c2	dir	system_low	1	downgraded dir
DSC-14:	D1	S,c1,c2	dir	S,c1,c2	1	(successful)
DSC-15:	D1	S,c1,c2	dir	S,c1,c2,c3	0	zero quota
DSC-16:	D1	S,c1,c2	seg	S,c1,c2,c3	-	upgraded seg
DSC-17:	[pd]	S,c1,c2	dir	S,c1,c2,c3	1	(successful)

DSC-7 to DSC-9: Upgrade in lower parent.

These three tests check to ensure that an upgraded directory cannot be created in a parent directory to which the process has "s" but no "m" permission due to the security controls. Both DSC-7 and DSC-9 would otherwise be legal. DSC-8 should be illegal anyway because the upgraded directory to be created has a lower category set.

DSC-10 and DSC-11: Upgrade in higher parent.

These two checks further verify that the lack of "m" permission inhibits creation of an upgraded directory. This time the parent directories are of a higher level and category set than the process.

DSC-12 and DSC-13: Downgraded directory.

These two tests verify that it is not possible to create a downgraded directory in which the access class of the directory is less than that of the parent. With respect to the current process authorization, both these attempts are otherwise legal. The purpose of DSC-13 is to check that system_low is not treated as a special case.

DSC-14: Directory of current authorization.

This test uses hcs_$create_branch_ to create a directory of the current authorization, and should be successful.

DSC-15: Upgraded directory with zero quota.

When calling hcs_$create_branch_, the caller specifies a quota for the directory to be created. It should not be legal to create an upgraded directory without quota, as attempted by this test.

DSC-16: Upgraded segment.

The hcs_$create_branch_ primitive allows the caller to create an upgraded segment, provided his validation level is in ring 1. This option is for use by the message segment software, which runs in ring 1. The user whose validation level is 4 should not be able to invoke this option.

DSC-17: Upgraded directory of higher access class.

Finally, this last test creates a valid upgraded directory of a higher access class than the current process. The directory is created in the process directory because otherwise it would be difficult to delete from the hierarchy. In order to verify that this directory is truly upgraded, try_dir_reference_ is called to check access to it.

DSC-18 to DSC-20: Implicitly modified attributes.

There are three final tests that check that the dtu and dtm of directories and segments are not implicitly modified when there is no modify permission (due to security controls) to the parent. In each of these tests, the access class of the parent of the directory or segment being referenced is less than the authorization of the process that might have caused the dtu or dtm to be modified. The table below lists the name of the directory or segment and its parent, whose dtu and dtm are checked.

	parent	name
DSC-18:	DIR_TEST	D1
DSC-19:	D2	DIR
DSC-20:	D2	SEG

In DSC-18, the dtu and dtm of D1, stored in the parent, should not be modified because the access class of D1 is greater than that of DIR_TEST and the access class of DIR_TEST is less than the process authorization. In DSC-19 the access class of the parent (D2) of DIR is equal to the access class of DIR, but lower than the process authorization. This test checks that the reason for not modifying the dtu was due to the authorization of the current process being "greater than" the access class of the parent directory -- not because the parent directory had a lower access class than the directory. DSC-20 checks that the dtu and dtm restrictions also apply to segments.

COMMUNICATION BETWEEN PROCESSES

Processes can communicate with each other by various means: segment or directory sharing, the interprocess communication facility (IPC), and message segments. Segment and directory sharing are automatically secure if the segment and directory controls work properly. IPC and message segments are special facilities that must be specifically tested. The design description and test procedures for each facility will be presented separately.

Interprocess Communication - Design Description

IPC is conceptually very simple: a process sends a message of fixed length to another process. With security controls, the authorization of the sending process is attached to the IPC message and becomes the access class of the message. A process can only receive a message if the authorization of the process is "greater than" or "equal to" the access class of the message.

Interprocess Communication - Test Procedures

IPC is tested in a straightforward manner by having processes of various authorizations send messages to one process having a fixed authorization. Since it is inconvenient to test with more than one or two processes (terminals) at a time, a scheme using two process is used. There is a sending process that starts at a given authorization and then changes its authorization using the new_proc command. At each unique authorization, it sends a message to a second receiving process. This receiving process remains at a fixed authorization.

The table below lists the six tests (IPC-1 to IPC-6) that are to be performed. The sending process is initially logged in at system_low, and the receiving process remains logged in at S,c1,c2. These six tests, each consisting of sending a single message of a given access class, are very similar to the segment and directory security controls tests. In fact, an exact correspondence with the tests DSC-1 to DSC-6 (see page 54) can be made, except that the sequence has been changed so that the "legal" situations come up first.

	P1	relation	results
IPC-1:	S,c1,c2	P1 = P2	Message received
IPC-2:	C,c1,c2	P1 < P2	Message received
IPC-3:	S,c1	P1 $\subset$ P2	Message received
IPC-4:	S,c1,c2,c3	P2 $\subset$ P1	Message not received
IPC-5:	T,c1,c2	P1 > P2	Message not received
IPC-6:	S,c1,c3	P1 $\not\leq$ P2	Message not received

In the above table P1 is the authorization of the first process, and P2 is the authorization of the second process, which stays fixed at S,c1,c2. The meanings of the other symbols are defined below the table on page 52.

Message Segments - Design Description

Message segments are special segments maintained by ring 1 software. A distinctive property of message segments is that they are multi-level. Message segments contain individual messages that may have been put there by processes of various authorizations. Each message has an access class associated with it, and access to the individual messages is subject to exactly the same security controls as is access to segments. There is also a special kind of need to know access control for messages involving the five access control bits:

a	add any message
d	delete any message
r	read any message
o	delete or read only own messages
s	obtain number of messages

The special ACL of message segments (called an extended ACL) is a list like that for regular segments, except that the five bits above are used instead of r, e and w.

When a user creates a message segment, usually for the purpose of receiving mail from other users, the ACL normally is set as follows:

```
adros     User.Project.*
ao        *.*.*
```

This ACL specifies that the creator of the message segment has full access to all messages, and that all other users have full access to their own messages.

When security controls are in force, the effective access mode to any particular message may be furtner restricted. The restrictions ensure that it is not possible to violate the *-property or the security condition. In particular, a message may not be deleted unless its access class is equal to the process authorization (and either "d" or "o" access is permitted in the extended ACL), and a message may not be read if its access class is "greater than" or "isolated from" the process authorization. This latter restriction also applies to learning of the existence of a message through the "s" access mode.

Because message segments are finite resources, it is possible for a message segment to fill up. When there is no more room in a message segment, the sender is notified, even if he has no access to any of the other messages in the segment. This makes it possible for a Trojan Horse to pass one bit of information (the "message segment overflow" condition) to another cooperating process, even if the second process is of a lower authorization. In order for this scheme to pass any significant amount of information, the second process must repeatedly cause the overflow condition to occur. There is no way to prevent such an occurrence in the current implementation without severely restricting the utility of message segments, so the solution was to audit such events in the hope that the penetrator would soon get caught. Under normal circumstances, this condition should occur infrequently enough to be easily distinguishable from a penetration attempt.

Message segments, as a whole, have an access class that is the maximum access class of any message that may be put into them. This value is set to the user's maximum authorization when the message segment is created. Enforcement of the message segment access class is not a requirement for security, since access to the individual messages is controlled. Its only purpose is to prevent message segments from containing messages that the user will never be able to read.

Message Segments - Test Procedures

Ideally message segments should be tested by invoking the ring 1 primitives that manipulate them. Unfortunately these ring 1 interfaces are considered internal to Multics and no documentation is generally available. In addition, they are subject to change. In order to provide a reasonable test of message segments that will remain generally useful in the future, the Multics mail facility is used. The mail command, along with several special mailbox commands, is a command level interface to the ring 1 primitives. For these tests it will be assumed that the user is not able to bypass any controls by invoking ring 1 directly.

There are again six tests of message segments similar to the six for IPC tests. The test procedure consists of creating a message segment and sending messages to it from processes at various authorizations. When all messages are sent, an attempt is made to access those messages from a process at a specific authorization in relation to the access classes of the messages.

The mailbox is initialized by a process that logs in at system_low, and creates a message segment using the mbx_create command. This process then new_procs itself to each of six authorizations and sends a message to this mailbox while at each authorization. When the six messages have been sent, the process new_procs to S,c1,c2 and attempts to read its messages. The mail command is not very specific with regard to individual messages -- The only options are to read all messages and to delete all messages. Thus, when the mail command is invoked, three of the messages to which the user has read permission should be printed, and the other three should not be. When the user attempts to delete the messages, only the one at access class S,c1,c2 should get deleted. The mail command is invoked again after deletion to check that only that one message got deleted.

The table below lists the access classes of the messages put into the mailbox. There are not actually six different "tests", since the mail command is invoked only twice -- once to read and delete all the messages, and a second time to check on the deletion. For consistency with IPC, however, this test will be listed as six tests. The symbol M is the access class of the message and P is the access class of the process. The other symbols have been defined earlier.

	M	relation	read?	deleted?
MBX-1:	S,c1,c2	$M = P$	yes	yes
MBX-2:	C,c1,c2	$M < P$	yes	no
MBX-3:	S,c1	$M \subsetneq P$	yes	no
MBX-4:	S,c1,c2,c3	$P \subsetneq M$	no	no
MBX-5:	T,c1,c2	$M > P$	no	no
MBX-6:	S,c1,c3	$M \neq P$	no	no

MBX-7: Deletion of mailbox.

In addition to the deletion of individual messages, the user normally has the ability to delete his entire mailbox. This deletion should not be allowed, however, if there are messages of an access class below his current authorization.[11] Of course, in order for a mailbox to contain messages of a lower authorization, that mailbox must be in a directory of a lower authorization; otherwise no process of a lower authorization could have known of the existence of the mailbox. Thus, the deletion of the mailbox would normally be subject to the usual rules for the deletion of segments in a directory of a lower access class. However, this test for mailboxes should be made because the deletion of a mailbox is handled by ring 1, which could bypass the security controls if it chooses. For this test, while the process is still at S,c1,c2, the user attempts to

[11]Note that it must be possible for the user to delete a mailbox containing messages of only higher and equal access classes. If deletion were restricted because of the presence of higher access class messages, the user could infer the existence of those messages by noting that the mail command tells him that there are no messages while at the same time he cannot delete the mailbox.

invoke the mbx_delete command to delete the mailbox. This deletion attempt should fail. Finally, if desired, the user can new_proc to system_low and delete the mailbox.

Note that there is no test of the enforcement of the maximum access class of the message segment, since this feature is not a requirement for security. Note also that these tests assume that the current user has "adros" access to the mailbox, which is the default condition when mailboxes are created.

ACCESS TO I/O

The area of input and output has traditionally been the most difficult to control in a secure manner. In Multics without security controls a process must "attach" a peripheral device, such as a tape drive or terminal, before that device can be accessed. This attachment can be viewed as similar to the act of initiation for a segment: the process' access privileges are determined at attachment time and access is allowed or denied. Since all I/O, like directory references, is ultimately performed by hardcore (unlike references to segments which are made directly by machine instructions), the attachment gives the user the right to access a particular device via the appropriate hardcore entries.

Because of the complexity of I/O, it was determined that bugs in hardcore I/O routines might exist that could be exploited by the user to bypass the security controls. Since validation of the hardcore I/O routines is not currently feasible, the decision was made to restrict attachment of devices (other than terminals) to system processes only. Any I/O that a user process wants to perform must be accomplished by submitting a request to a system process in some type of queue. Message segments, as described on page 58, are used to hold these queues and the user process' request is in the form of a message to the appropriate message segment.

The security controls require that the normal rules applying to segment accesses also apply to system precesses with respect to I/O device accesses. Therefore a process will only be able to use a device for writing if the authorization of the process is "less than" or "equal to" the access class of the device. Reading from a device is restricted to a process having an authorization "equal to" the access class of the device. The user's indirect access to the I/O device through the system process is also subject to the same controls. Following are the design descriptions and test procedures for the various I/O devices.

Card Input -- Design Description

Card decks submitted by the user are identified by two header cards: a user ID card and a deck ID card. The user ID card or cards contain the user's name, his project and the deck access class. The deck ID card contains the name of the deck and the name of the system process to be used for reading the deck. In addition a unique identifier card, supplied by the operator, is inserted before and after each card deck to ensure that each user deck is read separately. A card reader driver process reads in the card decks and places them into a segment in the card pool hierarchy. The driver process has no special privileges. Therefore all decks must have an access class identical to the authorization of the driver process. The driver process rejects all decks whose access classes are not identical. Although the driver is given no privileges with respect to the system security controls, it is trusted to refuse to read decks of the wrong access class.

The card pool hierarchy is a set of directories and segments containing the images of the card decks read. The root of the hierarchy is an unclassified card pool directory. Within the root is a directory for each access class currently required to store card decks. Within each access class directory is a directory corresponding to each user who has card decks in the pool. The actual card deck images are placed in segments within these user directories. To obtain a copy of the card deck image the user must copy the cards to one of his own segments. A garbage collector removes deck image segments from the card pool hierarchy at periodic intervals.

Card Input -- Test Procedures

The following tests check for the proper operation of the card input routines and related functions. In general since the I/O routines interpretively check the access class and ACLs, I/O routines must function correctly. Therefore the tests for all I/O devices include both security sensitivity tests and general operational correctness tests. At the beginning of these tests the card reader is logged into the system with an access class of secret,c1,c2.

CIF-1 to CIF-6: Security tests.

The first group of tests check for the proper operation of the card reader with regard to access classes. An attempt is made in these tests to input decks with different access classes to ensure that only decks with an access class equal to the access class of the card reader are read into the system. The relationships between the access class of the six decks (D) and the access class of the card reader (CR) are the same as those in the directory tests DSC-1 to

DSC-6, and are indicated in the table below.

	user.project	deck access class	relation	result
CIF-1:	u7.p5	S,c1,c2	D = CR	deck accepted
CIF-2:	u6.p5	U,c1,c2	D < CR	deck rejected
CIF-3:	u6.p5	S,c1	D ∈ CR	deck rejected
CIF-4:	u6.p5	S,c1,c2,c3	CR ∈ D	deck rejected
CIF-5:	u6.p5	T,c1,c2	D > CR	deck rejected
CIF-6:	u6.p5	S,c3	D ≰ CR	deck rejected

CIF-7: Unique ID Card test.

Card decks must be surrounded by identical unique identifiers. In this test a card deck is surrounded by unique identifiers that are different. The deck should be rejected and the operator notified.

CIF-8 to CIF-12: User ID card tests.

The following five tests check the validity of the user ID card. The first test, CIF-8, checks that the user ID card is properly read if the access class is expanded over more than one card. The second test of this group, CIF-9, ensures that a deck is rejected if, while the access class of the card reader is not unclassified, the access class field of the user ID card is omitted. Following test CIF-9 the access class of the card reader is changed to unclassified. The next test CIF-10 checks to ensure that a card deck with no access class on the user ID card is properly read while the access class of the card reader is unclassified. Test CIF-11 checks that an invalid access class on the user ID card is rejected by the card reader. The final test of this series, test CIF-12, checks that a * cannot be placed in the user field of the user ID card. The table below outlines the user ID card tests.

	user ID card(s)	results
CIF-8:	u7.p5 secret, c1,c2;	deck accepted
CIF-9:	u6.p5;	deck rejected
*** (Change level of card reader to unclassified) ***		
CIF-10:	u7.p5;	deck accepted
CIF-11:	u6.p5 unsecret;	deck rejected
CIF-12:	*.p5;	deck rejected

CIF-13 and CIF-14: Deck ID card tests.

These tests check the validity of the deck ID card. The tests continue to assume that the access class of the card reader is unclassified. On the deck ID card, the user is allowed to specify the name of a device interface module (DIM), which is the name of the program that will read and perform code conversion on his card deck. Test CIF-13 checks that a deck having a non-system DIM name on the deck ID card is rejected and the operator notified, thus ensuring that users can not specify their own card reading routines. In test CIF-14 a deck is read with the same user and deck ID cards as test CIF-10. This test ensures that decks with identical names are named differently in the system.

CIF-15 to CIF-23: Tests on results.

If tests CIF-1 to CIF-14 have been performed correctly four decks have been read into Multics through the card reader. The following tests check the results to ensure that the card reader routines are functioning properly. User u7 is used in all the seven tests.

Tests CIF-15 to CIF-18 ensure that the card decks read by the card reader are properly placed in the card pool hierarchy. In test CIF-15 the user lists the card pool directory to ensure that there are two directories: one corresponding to unclassified and one corresponding to secret,c1,c2. Test CIF-16 then lists the directory corresponding to secret,c1,c2 to ensure that there is a directory entry for user u7 and no entry for user u6. Test CIF-17 lists user u7's directory to ensure that there are two segments corresponding to the decks read in tests CIF-1 and CIF-8. The final test of this group CIF-18 prints the segment created by test CIF-1 to ensure that the deck has been read properly. Following is a summary of these tests.

	user.project	operation	result
CIF-15:	u7.p5	list card_pool directory	unclassified directory S,c1,c2 directory
CIF-16:	u7.p5	list S,c1,c2 directory	directory for u7
CIF-17:	u7.p5	list directory for u7	segment for test CIF-1 segment for test CIF-8
CIF-18:	u7.p5	print segment for test CIF-1	segment from test CIF-1

In tests CIF-19 to CIF-22 the access control lists of the directory and segments in the card pool are checked. Test CIF-19 checks the access control list for the card pool directory to ensure that no user has modify permission to the card pool. Test CIF-20 lists the access control list for the directory corresponding to secret,c1,c2 to ensure that no user has modify permission to this directory. Test CIF-21 checks the access control list for the directory created by user u7's card decks to ensure that only user u7 has status permission to this directory and to ensure that no user has modify permission. Finally test CIF-22 checks the access control list for the segment corresponding to the deck read in test CIF-1 to ensure that read permission for u7.p5 is the only user permission granted. The following table summarizes tests CIF-19 to CIF-22.

	user.project	ACL listed	result
CIF-19:	u7.p5	card pool directory	no user has modify
CIF-20:	u7.p5	S,c1,c2 directory	no user has modify
CIF-21:	u7.p5	u7's directory	only u7.p5 has status no user has modify
CIF-22:	u7.p5	segment corresponding to test CIF-1	u7.p5 has read no other user privileges

In the final test of this group, CIF-23, user u7 uses the copy_cards command to copy the unclassified file read in test CIF-10. User u7 should be notified of the existence of two copies of the file and the copy request should be properly performed.

CIF-24: Test of I/O attachment.

The security controls are effective only if attachment of devices is controlled by the operating system. Attachment of devices on Multics is done by calling the ioi_ primitive, which is a gate into ring zero. Test CIF-24 checks the access control list of ioi_ to ensure that no user may call ioi_$attach. Though this test is not strictly a card input test, it is performed here because it applies to all I/O devices.

Following these tests the operator should delete the directories in the card pool for user u7 so that future tests will perform properly.

Card Output -- Design Description

Card output, as well as printed output and most other I/O, is performed by a system process called an I/O daemon. (Card input is performed by a different type of daemon.) An I/O daemon is a system process that handles I/O requests. There are usually several I/O daemons logged into the system at any one time. There are two basic types of I/O daemons: the I/O coordinator, of which there is only one per system, and an I/O driver process, of which there is one per device. The I/O coordinator has special privileges with respect to security. The driver process have no special privileges other than the right to attach I/O devices.

To punch a deck a user sends a message to the I/O coordinator stating, among other things, the pathname of the segment to be punched. The I/O coordinator forwards the request to the driver process for the first available card punch that can handle the request.

The card punch driver can accept requests within a range of access classes. The maximum access class is the authorization at which the driver operates. The I/O coordinator only forwards requests to the drivers if the requests have an access class between the minimum and maximum access class associated with the device. The access class of a particular request is the authorization of the process that made the request -- not the access class of the segment to be punched.

To limit overclassification by the card punch an access class banner is punched with each deck. The access class banner is the least access class that is greater than or equal to both the authorization of the user process requesting the output and the minimum banner for the device.

Card Output -- Test Procedures

The following tests check for the proper operation of the card punch routines and related functions. Throughout these tests the card punch is assumed to have been logged in with the parameters specified at initialization. These parameters are as follows:

maximum access class	=	secret,c1,c2,c3,c4
minimum banner	=	restricted,c1,c2,c3
minimum access class	=	confidential,c1,c2

CPT-1 to CPT-6: Security tests.

The first group of tests for the card punch ensure that to punch a segment the process making the request must have an authorization in the range of the device. For each test a process of a different authorization attempts to punch an unclassified segment. In the table below, P is the authorization of the process and Min and Max are the minimum and maximum access classes of the device respectively.

	user.project	process authorization	relation	result
CPT-1:	u7.p5	C,c1,c2	P = Min	deck punched
CPT-2:	u7.p5	U,c1,c2	P < Min	no deck punched
CPT-3:	u7.p5	C,c1	P 6 Min	no deck punched
CPT-4:	u7.p5	C,c1,c2,c3,c4,c5	Max 6 P	no deck punched
CPT-5:	u7.p5	T,c1,c2	P > Max	no deck punched
CPT-6:	u7.p5	C,c2,c3	P ≰ Min	no deck punched

CPT-7 to CPT-9: Improper access checks.

Tests CPT-7 to CPT-9 check that the interpretive checks of the access class of the segment and the access control list are made correctly by the driver or I/O coordinator. Before punching it must be verified that the process that made the request had the proper authorization and was listed on the ACL of the segment to be punched. Normally, these checks are made at the time of request by the dpunch command in the user ring. In order to verify that the checks are made by the I/O daemon, a special version of the dpunch command is used for these tests that does not make any access checks before queuing the request.

In test CPT-7 a process with a confidential,c1,c2 authorization attempts to punch a segment with a secret,c1,c2 access class. The output should not be punched and the operator should be notified of the improper request.

In test CPT-8 a process with a confidential,c1,c2 authorization attempts to punch a segment with a confidential,c1,c2,c3 access class. The segment should not be punched and the operator should be notified of an improper request.

For test CPT-9 an attempt is made to punch a segment to which the user does not have read access on the ACL. The segment should not be punched and an error message should be produced.

CPT-10 and CPT-11: Banner checks.

Tests CPT-10 and CPT-11 check the banner for the card punch. Test CPT-10 tests the minimum banner of the device. Test CPT-11 ensures that, if the access class of the process making the request is greater than the minimum banner, the authorization of the process is used as the banner.

	user.project	process authorization	banner
CPT-10:	u7.p5	C,c1,c2	R,c1,c2,c3
CPT-11:	u7.p5	S,c1,c2,c3,c4	S,c1,c2,c3,c4

Following these tests the operator should clear the queues of the requests made in tests CPT-2 through CPT-6.

Printed Output -- Design Description

Local and remote line printers, like card punches, are run by system I/O daemons. Each printer has a maximum access class that is the maximum access class of data that can be printed, a minimum access class that is the minimum authorization of a process that can request data to be printed on that printer, and a minimum banner that is the minimum access class name appearing in block letters on the first page of output of each segment printed. Along with each printer, there is an accountability terminal that is used to print an accountability form[12] for each segment printed on the printer.

In addition to the security controls mentioned, the user has the ability to print access class labels at the head and foot of each page of output. These labels cannot be trusted to display correctly the access class of the data since the user can change them. However, they do provide a framework for per page classification.

Printed Output -- Test Procedures

The following tests check for the proper operation of the printer routines and related functions. The first eleven tests are identical to the tests performed for the card punch. The two printers prt1 and prt2 are assumed to be initialized as indicated in the table on page 34. For tests LPT-1 to LPT-16 and tests LPT-20 to LPT-22 a single printer prt1 is used having the following parameters:

maximum access class	= secret,c1,c2,c3,c4
minimum banner	= restricted,c1,c2,c3
minimum access class	= confidential,c1,c2

Tests LPT-17 to LPT-19 require both printers. The second printer, prt2, has the following parameters:

maximum access class	= top secret,c1,c2,c3,c4,c5
minimum banner	= top secret,c1,c2,c3,c4
minimum access class	= secret,c1,c2,c3,c4

LPT-1 to LPT-6: Security tests.

The first group of tests for the printer ensure that to print a segment a process must have an authorization in the range specified for the printer. For each test a process of a different authorization attempts to print an unclassified segment. The table below outlines the security tests.

[12]Accountability forms, e.g. AF form 310, are required by the military for each classified document produced.

	user.project	process authorization	relation	result
LPT-1:	u7.p5	C,c1,c2	P = Min	segment printed
LPT-2:	u7.p5	U,c1,c2	P < Min	no segment printed
LPT-3:	u7.p5	C,c1	P ⋦ Min	no segment printed
LPT-4:	u7.p5	C,c1,c2,c3,c4,c5	Max ⋦ P	no segment printed
LPT-5:	u7.p5	T,c1,c2	P > Max	no segment printed
LPT-6:	u7.p5	C,c2,c3	P ≰ Min	no segment printed

LPT-7 to LPT-9: Improper access checks.

Tests LPT-7 to LPT-9 check that the printer driver and I/O coordinator perform the interpretive checks for the access class of the segment and access control list correctly. Before printing it must be ensured that the process that requested the output had the proper authorization. As for the card punch tests, a special version of the dprint command is used that does not check for proper access before queuing the request.

In test LPT-7 a process with a confidential,c1,c2 authorization attempts to print a segment with a secret,c1,c2 access class. The output should not be produced and the operator should be notified of the improper request.

In test LPT-8 a process with a confidential,c1,c2 authorization attempts to print a segment with a confidential,c1,c2,c3 access class. The segment should not be printed and the operator should be notified.

In test LPT-9 an attempt is made to print a segment to which the user does not have read access. The segment should not be printed, but instead an error message should be produced.

LPT-10 and LPT-11: Banner checks.

Tests LPT-10 and LPT-11 check the banner for the printer. Test LPT-10 tests the minimum banner of the device. Test LPT-11 ensures that the process authorization is used as the banner in the case where the process making the request has an authorization greater than the minimum banner.

	user.project	process authorization	banner
LPT-10:	u7.p5	C,c1,c2	R,c1,c2,c3
LPT-11:	u7.p5	S,c1,c2,c3,c4	S,c1,c2,c3,c4

LPT-12 to LPT-16: Header and footer tests.

Each page of printed output has page label fields which, unlike the banners, are placed at the discretion of the user. The fields consist of character strings appearing at the top and bottom of each page of printout. The user can either explicitly specify the contents of these labels, or he can specify as a default that the labels indicate the access class of the segment. Although these discretionary labels cannot be trusted, users may rely on them to display correctly the access class of the data. Thus, tests for the correct functioning of the page labels are necessary.

Test LPT-12 checks the default label option to ensure the segment's access class is used for the header and footer label for each page of output. Test LPT-13 checks the access class option to ensure that the access class of the segment is used for the header and footer labels. Test LPT-14 checks the label option to ensure that the user-supplied character string is used as the header and footer label for each page of printed output. LPT-15 checks the top label option to ensure that the user supplied character string appears only on the top of each page while the bottom label is the segment's access class. The corresponding bottom label option is tested in test LPT-16. Following is a summary of the header and footer tests.

	option	top label	bottom label
LPT-12:	default	segment access class	segment access class
LPT-13:	-access_class	segment access class	segment access class
LPT-14:	-label	user supplied label	user supplied label
LPT-15:	-top_label	user supplied label	segment access class
LPT-16:	-bottom_label	segment access class	user supplied label

LPT-17 to LPT-19: Queue group tests.

This group of tests checks the proper operation of queue groups. A queue group is a collection of devices that share a queue of requests. Since each request in a queue may have a different access class, it is necessary to check that each request in a queue is sent only to the device that can accept it according to the device's access class range. The queue groups are used for other devices besides printers, but the test is made only for printers because the software in the I/O coordinator is identical for all device types.

A second printer initialized as prt2 is needed for these tests. Test LPT-17 tests the proper operation of the queue group by submitting two simultaneous dprint requests thus forcing a segment to print on each printer. Test LPT-18 ensures that if a request has a level greater than that of one device in the queue it will be printed by another device in the queue having the proper level. This test is constructed so that, if level checks were improperly made or ignored, the segment would be printed on the wrong printer. Test LPT-19 is identical to test LPT-18 except that the category set is greater rather than the level.

LPT-20 to LPT-22: Accountability terminal tests.

With each printed output an accountability form is printed. This form is used to interface the computer's internal security controls with the external environment. The accountability form contains pertinent information regarding security and must be checked for correctness. Test LPT-20 checks the correctness of each accountability form produced in the previous tests. The checks include correctness checks for the proper user name, user project, proper date and time, pathname of the segment printed, sequence number, and the access class of the segment. Test LPT-21 ensures that a printer requiring an accountability terminal will not perform output if its accountability terminal is not dialed up. The final test LPT-22 ensures that, should an accountability form terminal be disconnected during printing, the printer will cease to accept output requests. This feature guarantees that the number of forms printed will be equal to the number of requests printed.

Following the completion of the above tests the operator should delete any requests remaining in the queues as a result of tests LPT-2 to LPT-6 and LPT-22.

Tape I/O -- Design Description

Magnetic tape input and output is also performed by I/O daemons. Currently however, the majority of the security controls are manually performed by the operator. Future releases of the system are planned to integrate some of the security controls into the system to simplify the operator's interface.

All magnetic tapes used on the system are registered and assigned an access class. When a user desires to read or write a tape he executes a read_tape or write_tape command. The command adds the request to a queue and notifies the operator of the request. The operator must assign the tape drive, mount the tape, verify that the user has the need-to-know to read or write the tape, and verify that the user's authorization allows him the requested access. The operator assigns the access class of the drive to be the same as the access class of the tape. Tapes may be read into a specified segment or into a tape pool hierarchy, similar to the card pool hierarchy. Any segment that the user can read may be written onto a tape.

One disadvantage with the current system is that there is no means by which the system can differentiate between tape drives assigned for reading and those assigned for writing. The result of this restriction is that the minimum and maximum access classes of the tape daemon must be identical. The operator must ensure that this requirement is met. The minimum banner parameter has no effect on tape drives.

<u>Tape I/O -- Test Procedures</u>

The following tests check for the proper operation of the tape input and output routines and related functions. The tape pool hierarchy is not tested here as it is identical to the card pool hierarchy tested in card input and is managed by the same software. In addition, only the software is tested. Checks performed by the operator as described above are not tested. The tests TDT-1 to TDT-9 below apply to tape output, and TDT-10 and TDT-18 are for tape input. Throughout these tests the magnetic tape drive is assumed to have been initialized for reading and writing at the access class confidential,c1,c2.

TDT-1 to TDT-6: Security tests for writing a tape.

The first group of tests for tape output ensures that the process requesting the writing of a tape has an authorization equal to that of the device. For each test a process of a different authorization attempts to write an unclassified segment onto tape. In the table below, TD is the access class of the tape drive.

	user.project	process authorization	relation	result
TDT-1:	u7.p5	C,c1,c2	P = TD	tape written
TDT-2:	u7.p5	U,c1,c2	P < TD	no tape written
TDT-3:	u7.p5	C,c1	P $\subset$ TD	no tape written
TDT-4:	u7.p5	C,c1,c2,c3,c4,c5	TD $\subset$ P	no tape written
TDT-5:	u7.p5	T,c1,c2	P > TD	no tape written
TDT-6:	u7.p5	C,c2,c3	P ≠ TD	no tape written

Note again that there is no check for the access class of the tape itself. That must be verified by the operator.

TDT-7 to TDT-9: Improper access checks for writing a tape.

Tests TDT-7 to TDT-9 check that the tape driver and the I/O coordinator perform the interpretive check for the access class of the segment and access control list correctly. Before writing a tape it must be ensured that the process requesting the writing of the tape has an authorization equal to the access class of the tape drive and driver process. It must also be verified that the access control list of the segment to be read allows the requestor access to the segment. For these tests, a special version of the write_tape command is used that does not check for access, but submits the request in all cases.

In test TDT-7 a process with a confidential,c1,c2 authorization attempts to write onto a confidential,c1,c2 tape a segment with a secret,c1,c2 access class. The tape should not be written and the operator should be notified of the improper request.

In test TDT-8 a process with a confidential,c1,c2 authorization attempts to write onto a confidential,c1,c2 tape a segment with a confidential,c1,c2,c3 access class. The tape should not be written and the operator should be notified.

In test TDT-9 an attempt is made to write onto a tape a segment to which the user does not have read access. The segment should not be written and an error message should be produced.

TDT-10 to TDT-15: Security tests for reading a tape.

The first group of tests verifies that to read a tape the process making the request must have an authorization equal to that of the tape drive. For each test a process of a different authorization attempts to read a tape. The table below outlines the security tests.

	user.project	process authorization	relation	result
TDT-10:	u7.p5	C,c1,c2	P = TD	tape read
TDT-11:	u7.p5	U,c1,c2	P < TD	no tape read
TDT-12:	u7.p5	C,c1	P ⊂ TD	no tape read
TDT-13:	u7.p5	C,c1,c2,c3,c4,c5	TD ⊂ P	no tape read
TDT-14:	u7.p5	T,c1,c2	P > TD	no tape read
TDT-15:	u7.p5	C,c2,c3	P ≠ TD	no tape read

TDT-16 to TDT-18: Improper access checks for the tape reader.

Tests TDT-16 to TDT-18 verify that the interpretive access class and access control list checks are made properly, in a manner similar to that for the other devices.

In test TDT-16 a process with a confidential,c1,c2 authorization attempts to read a tape into a secret,c1,c2 access class segment. The operator should be notified of the improper request.

In test TDT-17 a process with a confidential,c1,c2 authorization attempts to read a tape into a confidential,c1,c2,c3 segment. The operator should again be notified.

In test TDT-18 a process not having write access to a segment attempts to read a tape into that segment. An error message should be produced and no tape should be read.

TDT-19: Check of data read and written.

For this test the segment created as a result of reading the tape in test TDT-10 is printed. This test ensures that both the write_tape command (which originally was used to write the tape in TDT-1) and read_tape command are functioning correctly.

Following these tests the operator should delete from the queues requests generated as a result of tests TDT-2 through TDT-6 and TDT-11 through TDT-15.

SYSTEM SECURITY ADMINISTRATOR

Design Description

The System Administrator is an individual who has access to certain administrative commands and data bases in Multics, such as the PNT, SAT, etc. The normal System Administrator can carry out his functions without regard to security controls, and his functions are not affected by the addition of security controls. The System Security Administrator is the only individual who is permitted to perform certain security related operations within the hierarchy, such as the reclassification of segments and directories, that may occasionally be required. Because of the sensitive nature of his operations, the SSA is only given a special limited subset of available Multics commands required to perform his functions (see the discussion of system processes on page 19). He is not permitted to call most of the user commands, nor can he arbitrarily invoke other users' programs. The complete set of commands available to the SSA is determined by the installation, but there are certain commands that have been especially designed for use by SSAs. These are:

reclassify_dir	upgrade/downgrade a directory
reclassify_seg	upgrade/downgrade a segment
reclassify_sys_seg	upgrade/downgrade a ring 1 segment
reset_soos	reset security out of service bit
set_system_priv	set/reset system privilege bits

In order to use any of these commands, and to perform certain other functions that may be illegal for the average user, the SSA is given access to a special "system_privilege_" gate into hardcore that is used to set or reset certain per process privilege bits. Each privilege bit allows the current process to bypass the security controls when performing operations in a certain area. There are five privilege bits:

dir	directory privilege
seg	segment privilege
ipc	IPC privilege
ring1	ring 1 privilege
soos	security out of service privilege

The first three bits -- dir, seg, ipc -- can be set to bypass all security related checks with respect to directories, segments, and IPC. For example, with the directory privilege set, the user can list the contents of directories of a higher access class than the current authorization. With the IPC privilege, IPC messages can be received and sent to processes of all authorizations. The ring1 privilege bit specifies that security checks performed in ring 1 subsystems be bypassed. The soos privilege bit causes the security out of service flag to be ignored when the process performs operations on directories and segments.

The security out of service bit is set in a directory or segment whenever the system detects an inconsistency in the hierarchy. One example is a directory whose access class is not greater than or equal to the access class of its parent. When the security out of service bit is set, the directory or segment cannot be accessed by any process until the bit is reset, except for the special case of a process with the soos privilege bit set.

The first four SSA commands listed above are used to perform "repair" type operations in the hierarchy. The set_system_priv command can explicitly set or reset any of the five system privilege bits, so that certain "normal" user commands, such as delete, move_quota, etc., can be used by the SSA without interference from the security controls. The set of "normal" user commands available to the SSA should be small to minimize the possibility of human error or a Trojan Horse resulting in a compromise of security.

Test Procedures

In order to be reasonably certain that the commands used by the SSA work as expected, it would be necessary to test every command available to him. This is not possible, however, since his set of commands is not explicitly defined -- any Multics command could potentially be available to the SSA if the installation chooses. It is left up to the installation to make sure that commands available to the SSA work properly and have no Trojan Horses. The only SSA tests that are specified here are the tests of the five SSA commands and the proper enforcement of the five privilege bits.

SSA-1: ACL of system_privilege_ gate.

Probably the most important test is a check that only the SSA has access to the system_privilege_ gate. This test consists of listing the ACL of system_privilege_, and checking to make sure that only the SSA and perhaps SysDaemons have execute access. The SSA himself could make this check.

SSA-2 to SSA-7: Check of set_system_priv and enforcement of privilege bits.

These tests check that each privilege bit allows only that specified privilege and no others, and that the set_system_priv command properly sets or resets the privilege desired. For each of these tests a certain configuration of system_privilege bits is set, and a program is called that attempts to perform four operations, each one of which is illegal unless the appropriate privilege bit (dir, seg, ring1 or soos) is set. Test of ipc privilege is made by manually attempting to send a message to another user logged in at a lower authorization. The table below lists the arguments passed to set_system_priv, and the resulting state of the privilege bits, for each test.

	set_system_priv	privilege bits
SSA-2:	(none)	(none)
SSA-3:	dir	dir
SSA-4:	^dir seg	seg
SSA-5:	^seg ipc	ipc
SSA-6:	^ipc ring1	ring1
SSA-7:	^ring1 soos	soos
SSA-8:	^soos	(none)

SSA-9 to SSA-12: reclassify_dir, reclassify_seg, reclassify_sys_seg, reset_soos checks.

These four tests verify that the four commands perform as expected. No attempt is made to thoroughly test all possible cases, but the usual uses of the commands are checked. For example, the three reclassify commands are checked by reclassifying the objects and then listing their access classes. The reset_soos command is checked by first creating an inconsistency in the hierarchy and thus causing the security out of service bit to be set on a directory. Then reset_soos is invoked to see that the bit is not reset until the inconsistency is corrected.

SSA-9: reclassify_sys_seg
SSA-10: reclassify_seg
SSA-11: reclassify_dir
SSA-12: reset_soos

AUDITING

Auditing is a feature of Multics not unique to the security controls. For example, illegal logins, system errors, etc., have always been audited. With the addition of security controls, and because of Department of Defense requirements, certain "normal" events involving classified data are audited. In addition, other "abnormal" events (faults, access violations) are also audited, some to detect possible penetration attempts as discussed in Section II. For these test procedures we will only be concerned with the "protection audit" mechanism that involves the additional auditing features incorporated with the security enhancements. Auditing of illegal logins can be easily checked by examining the audit log after running the login tests and password validation tests (PAA and PDS series).

Design Description

The protection audit mechanism is designed to audit certain events for specific users. The System Administrator has a command that can set certain audit bits for any given user or all users on any given project. These bits specify which events are to be audited in processes created for the user or for any user on the project to be audited.

There are 13 audit bits as follows:

seg_init	Segment initiations
dir_init	Directory initiations
mc_seg_init	Message segment initiations
no_access	Access denied
ipr_fault	Illegal procedure faults
acv_mode	Mode access violations
acv_ring	Ring access violations
no_wakeup	Wakeup denied
sys_priv	Setting/resetting of system privileges

ssa_ops	Reclassifications
no_attach	Device attachment denied
no_mount	RCP mount denied
mseg	Message segment overflow

All audited events are inserted into a file called the syserr_log. The print_syserr_log command can be used by the System Administrator to select certain kinds of information out of this log.

The audited events fall into several classes:

1) Events that are legal and normal, but that must be audited for accountability (seg_init, dir_init, mc_seg_init).

2) Events that are unusual but cannot directly be exploited if the system functions properly (no_access, ipr_fault, acv_mode, acv_ring, no_wakeup, no_attach, no_mount).

3) Events that are unusual and can be exploited (mseg).

4) Events that should only occur for the SSA (ssa_ops).

5) Events that should only occur for system processes (SSA, SysDaemons, etc.).

If the system is functioning properly under normal circumstances, there will be many events in group 2, and thus it would probably only make sense to set these bits on for certain "suspect" users or projects. In addition, if there is a great deal of classified processing, there will be many events in group 1 (the auditing of initiations only applies to objects not at system_low). Events in groups 3, 4 and 5 should probably be audited for all users, including the SSA (but not SysDaemons). Any messages in group 4 or 5 can indicate that someone has obtained access to the system_privilege_ gate or obtained the password of the SSA. Messages in group 3, under normal circumstances, should occur very infrequently. Many events in this group (caused by a single user) indicates the possibility of a penetration attempt.

Test Procedures

Auditing is tested in a straightforward manner. All audit bits are set on for a given user, and the user performs a series of operations designed to trigger each of the auditing functions. There is no test of the command used by the SSA to set or reset the audit bits. It is assumed that this command works properly, and that each audit bit applies only to that one auditing function and no others. In order to fully test auditing the user must have access to the system_privilege_ gate so that the ssa_ops audit function can be tested. Thus, the SSA himself will probably perform the tests below. In addition, the user must be logged in at an authorization above system_low.

AUD-1 to AUD-3: seg_init, dir_init, mc_seg_init

The first three audit bits are tested simply by initiating a segment, directory, or message segment (mailbox). Since directories cannot be directly initiated by the user, directory initiation is forced by some reference to the directory.

AUD-4: no_access

This auditing function is tested by attempting to get the status of a directory to which the user has no access.

AUD-5 to AUD-7: ipr_fault, acv_mode, acv_ring

These three auditing functions are invoked when the user causes a fault to occur by executing an illegal machine instruction attempting an illegal access due to mode or ring bracket restrictions. To test these functions the user calls a program that attempts such operations.

AUD-8: no_wakeup

In order to test this function a user must login at another terminal with a lower authorization than the current user. The current user then attempts to use the Multics send_message command to send that user a message. The wakeup signal that normally occurs should not get transmitted (which was already tested by the IPC tests) and the event should be audited.

AUD-9 and AUD-10: sys_priv, ssa_ops

The ssa_ops bit refers to the reclassify commands used by the SSA. To test these functions, the user calls set_system_priv and one of the reclassify commands.

AUD-11 and AUD-12: no_attach, no_mount

These two auditing functions are invoked by attempting an attach or mount to any I/O device. Since only SysDaemons are allowed to attach or mount I/O devices, these attempts should be audited.

AUD-13: mseg

To test the mseg function, the max length of the user´s mailbox is temporarily set very small, and a large message is sent to it. A mailbox overflow occurs and the event should be audited.

After performing the above tests, the SSA (who may also be the user making the tests) uses the print_syserr_log command to print all protection audit messages that occurred since the beginning of the audit tests. The correspondence between the message in the syserr log and the command executed can then be verified.

SECTION IV

CONCLUSION

The security test procedures are designed to test that the security enhancements to Multics perform as required with respect to authorization and access class controls. The areas tested are those of password distribution, process authorization assignment, segment and directory access, communication between processes, I/O, auditing, and system security administrator functions.

Although the test procedures often try to "subvert" the system by attempting illegal operations, no amount of testing of a system that is not formally validated will guarantee that the security controls cannot be bypassed. The purpose of testing is to give reasonable assurance that the security controls are invoked where expected, and that the controls function as expected. This assurance is important because new system releases are issued several times a year. A typographical error or oversight in coding of security related software should be detected by the test programs -- an obscure design deficiency allowing some peculiar bypass of the controls will probably not.

APPENDIX I

TEST ENVIRONMENT INITIALIZATION

The procedure for initialization of the test environment is described in this appendix. Refer to Section III, beginning on page 26, for a discussion of this initialization.

USERS, PROJECTS, AND TERMINALS

The initialization of these components of the test environment was described in detail in Section III. This initialization requires the SA, SSA, and perhaps a user designated as the project administrator for the test projects (who may be the SA). The exact sequence of commands required to perform this initialization is not given here because of the numerous variations likely to be encountered. Rather, the specific attributes of the users and projects as specified in the table on page 29 are reproduced here for convenience.

PNT	PDT for p1 & p3	SAT	CDT
u1=C	u1=S	p1=S	t1=C
u2=T	u2=T	p2=C	t2=T
u3=S	u3=U	p3=C,1,2,3,4,6,7	t3=C,1,3,5,6,7
u4=C,1,3,4,5,6,7	u4=C,1,2,4,5,6,7	p4=C,4,5,6	t4=C,1,2,3,4,5,6,7
u5=C,1,3,4,5,6,7	u5=C,1,2,4,5,6,7	p5=system_high	t5=system_high
u6=system_high	u6=none		
u7=system_high	u7=none		

It is assumed that the SA and SSA are familiar enough with the registration of new users and projects so that the exact procedure is obvious. Attributes of the users, projects and terminals not specified in the table above are set to the default values. The only exception is user u2, whose PNT entry must specify that u2 cannot use the change_password option in his login line. The "generate_password" attribute should be set for u2.

It may be that at the installation there are not normally five terminals available with the above authorizations. In this case it may be necessary for the SSA to set up five terminals with the above authorizations each time the tests are run. Actually, the specific authorizations of the terminals, projects and users are only important during the PAA tests. For other tests, any terminals, projects or users may be used that have a maximum authorization sufficient to perform the tests. The table below lists the maximum authorizations of users required for each of the other test series.

PDS	unclassified
SAC	unclassified
SSC	secret,c1,c2
DSC	secret,c1,c2
IPC	top secret,c1,c2,c3
MBX	top secret,c1,c2,c3
CIF	top secret,c1,c2,c3
CPT	top secret,c1,c2,c3,c4,c5
LPT	top secret,c1,c2,c3,c4,c5
TDT	top secret,c1,c2,c3,c4,c5
SSA	(user must be SSA)
AUD	(user must be SSA)

Thus, except for the PAA tests which require users with specific maximum authorizations, the other tests may be performed by any users able to login at the authorizations shown in the table. In the test scripts in Appendix II, users u6 and u7, project p5, and terminal t5 are used in the examples because they all have an authorization of system_high. For the PAA tests, users u1 - u5, projects p1 - p4, and terminals t1 - t4 are used.

DIRECTORIES AND SEGMENTS

There are five special subtrees required for the tests. The first three, required for the authorization_tester, segment security controls tests, and directory tests, are created by exec_coms. The fourth subtree for the I/O tests is created manually. Any user with a maximum authorization of system_high can create these subtrees. The fifth subtree, for the SSA tests, must be created by the SSA. In the series of steps that follows, it is assumed that the user is initially logged in at system_low and that the full set of commands and subroutines for the test procedures are available in the user's search rules.

1. Initialize start_up.ec

In the start_up.ec in the user's home directory, the following line should be inserted to be executed at new_proc, but not login time:

```
&if [equal &1 new_proc] &then exec_com create_test_start_up
```

If the user is not in the process of creating test directories, execution of this line at new_proc should have no effect. Documentation on this exec_com can be found in Appendix III.

In addition, for the IPC tests, user u6 must call the test_ipc command in his start_up.ec at new_proc time. The following command line should be inserted:

```
&if [equal &1 new_proc] &then test_ipc -go
```

If the user is not in the process of running the IPC tests, execution of this command should have no effect. Documentation on the test_ipc command can be found in Appendix III.

2. Initialization of ACL for authorization_tester subtree

Before creating any directories, it must be determined who is to be on the ACLs. The directories for the authorization_tester should at least have all users u1 through u5 under projects p1 through p4 on the ACL. However, since the only access modes granted to users on the ACLs of the directories and segments in this subtree are r and s, there is no harm in putting *.*.* on the ACL. Once the names of the users on the ACLs have been determined, a segment called "create_test_acl" should be created in the home directory. This segment must contain one group identifier per line as it is specified for an ACL, but without any access mode. For the authorization_tester subtree, it is sufficient to include one line in the segment containing "*.*.*".

3. Directory for authorization_tester

Decide on a pathname for the parent directory of the authorization tester subtree as illustrated in Figure 2 and move sufficient non-zero quota to it. The minimum amount of quota depends on the number of levels and categories within system_high. The exec_com "create_test_auth.ec" should then be called, with arguments as specified in the writeup of that exec_com in Appendix III. As an example, the following command line will create the proper subtree for an installation where system_high is T,c1,c2,c3,c4,c5,c6,c7:

```
exec_com create_test_auth AUTH_TEST "(C R S T c1 c2 c3 c4 c5 c6 c7)"
```

The above command line will create the authorization_tester subtree in the working directory, with the name PARENT. Creation of this subtree will take quite a while, since the process must new_proc to each level and category specified. Each new_proc will result in a message being printed naming the current authorization. When the command is completed, the user will be returned to system_low in his original working directory.

4. Initialization of ACL for directory and segment test subtrees

The create_test_acl segment in the home directory (initialized in step 2 above) should now be changed to contain a list of the group identifiers of the users who will perform the directory and segment security controls tests (DSC and SSC series). The group id of *.*.* should <u>not</u> be used, since modify permission will be given to the users specified. Probably an identifier like *.SysMaint.* and *.SysAdmin.* should be used. The scripts in Appendix II use u7 on project p5 when running the DSC and SSC tests.

5. Directory for segment and directory tests

Decide on a pathname for the two directories DIR_TEST and SEG_TEST, as illustrated in Figures 3 and 4. Move sufficient quota to the parents of these directories, and execute the following two commands:

```
exec_com create_test_dir DIR_TEST -acc C,1,2 T,1,2 S,1,2 S,1
     S,1,2,3 S,1,3
exec_com create_test_seg SEG_TEST -acc C,1,2 T,1,2 S,1,2 S,1
     S,1,2,3 S,1,3
```

Again, the above are just examples of valid calls to the commands. Refer to the writeups of create_test_dir.ec and create_test_seg.ec in Appendix III for a description of the various arguments.

6. Directory for I/O tests

Decide on a pathname for the directory IO_TEST, as illustrated in Figure 5. The following script shows how to create the IO_TEST subtree. The segment named 10_pages is the pathname of any segment able to generate at least ten pages of printed output. Ready messages have been omitted.

```
create_dir IO_TEST -quota 20


change_wdir IO_TEST

qx
a
This segment cannot be read due to a bad read ACL.
\f
w BAD_ACL
c
This segment contains proper information for output.
\f
```

```
w OUTPUT
c
This segment should not produce any output.
\f
w NO_OUTPUT
q

create_dir C,1,2 -access_class confidential,c1,c2 -quota 1

create_dir C,1,2,3 -access_class confidential,c1,c2,c3 -quota 1

create_dir S,1,2 -access_class secret,c1,c2 -quota 1

copy 10_pages LONG

set_acl * r *.p5.* -sm

delete_acl BAD_ACL -all

set_acl BAD_ACL we *.p5.*

set_acl C,1,2 sma u7.p5.*

new_proc -auth confidential,c1,c2

change_wdir IO_TEST>C,1,2

qx
a
This segment cannot be written due to a bad write ACL
\f
w BAD_WRITE_ACL
q

delete_acl BAD_WRITE_ACL -all

set_acl BAD_WRITE_ACL re *.*.*

new_proc -auth confidential,c1,c2,c3

change_wdir IO_TEST>C,1,2,3

qx
a
This segment should not be placed on output due to the
process' inability to read this segment due to categories.
\f
w BAD_CATEGORY
```

```
q

set_acl BAD_CATEGORY re *.p5.*

new_proc -auth secret,c1,c2

change_wdir IO_TEST>S,1,2

qx
a
This segment should not be placed on output due to the
process' inability to read this segment due to levels.
\f
w BAD_LEVEL
q

set_acl BAD_LEVEL re *.p5.*
```

7. Directory for SSA tests

Choose a pathname for the directory SSA_TEST as illustrated in Figure 6. The subtree must be created by the SSA logged in at the required authorization. The following script is an example of the creation of this subtree.

```
login SSA

create_dir SSA_TEST -quota 3 -access_class secret

new_proc -auth secret

change_wdir SSA_TEST

create SEG

create_dir DIR

reclassify_dir DIR "top secret"

set_system_priv soos

reclassify_dir DIR secret

mbx_create MSEG

logout
```

The sequence of steps above creates the upgraded directory SSA_TEST at the access class secret, and the contained segment, message segment, and out of service directory. Since there is no direct way to set the out of service bit, the out of service bit is set by upgrading the directory without quota, and then downgrading it again. The system privilege "soos" must be used to reclassify an out of service directory.

I/O DEVICES

There are five devices required for the I/O tests: a card reader, a card punch, two line printers with accountability terminals, and a magnetic tape input/output device. The devices used in the I/O tests must have specific values of the minimum access class, maximum access class, and minimum banner parameters. Since these values are not likely to be the same values normally used by the installation, special device types should be created in the I/O daemon parms file that contain the proper values of the parameters. Then, before performing a test of a particular device, the proper I/O driver can be logged in with the proper device type specified. The following table, reproduced from Section III, shows the values of these parameters.

device	name	min class	max class	min banner
card reader	crd	(none)	S,c1,c2	(none)
card reader	crd	(none)	U	(none)
card punch	punch	C,c1,c2	S,c1,c2,c3,c4	R,c1,c2,c3
line printer	prt1	C,c1,c2	S,c1,c2,c3,c4	R,c1,c2,c3
line printer	prt2	S,c1,c2,c3,c4	T,c1,c2,c3,c4,c5	T,c1,c2,c3,c4
tape drive	tape	C,c1,c2	C,c1,c2	(none)

APPENDIX II

TEST SCRIPTS

This appendix contains the scripts for the tests discussed in Section III. For the test sequences consisting entirely of scripts, the complete test scripts are shown. For the test sequences that are composed mostly of programs, the script shows the login and the call to the program performing the test. In cases where a program performs more than one test, a range of test numbers (e.g. "SSC-3 to SSC-10") is indicated for a given command line.

The scripts are examples of what the printout at a terminal might look like for each test. They are not to be taken literally in all cases. In particular the names of users, classifications, and contents of certain messages are shown only as examples. Lines totally or partially containing input typed by the user are preceded by an asterisk (*). Lines in parentheses are comments or instructions. All other lines are produced by the computer. An indented line may be considered a continuation of the previous line. Each test beginning with a test number can be run independently of the others, except when CONTINUE appears as the first line of a script. The tests marked CONTINUE are dependent on the preceding test and therefore must be run immediately following.

Command lines that contain calls to commands or active functions provided as part of the test package are indicated in the scripts by the symbol ">" in the left margin. Explicit usage of these commands is documented in Appendix III, and the source listings of many of these appear in Appendix IV in alphabetical order. All other commands are standard Multics commands documented in the Multics Programmers' Manual [10].

PASSWORD DISTRIBUTION (PDS)

The scripts of the password distribution tests described in Section III (beginning at page 34) are presented below. The scripts are numbered PDS-1 through PDS-7, corresponding to the test numbers in the text of Section III. The project referenced in these tests is set up in the test environment discussed on page 26 and detailed in Appendix I.

```
PDS-1:  (SA logs in and initializes password of u1 to "pass1".)
        (from terminal t1)
         Multics 1.1.1: AF Data Services Center.
         Load = 27.0 out of 50.0 units: users = 27
        *login u1
         Password:
        *xxxxx (This should be u1´s old password.)
         Login incorrect.
         Please try again or type "help" for instructions.

PDS-2:  (CONTINUE)
        (User puts terminal back online)
         Multics 1.1.1: AF Data Services Center.
         Load = 27.0 out of 50.0 units: users = 27
        *login ul-change_password
         Password:
        *pass1
         New password:
        *arptoa
         Password changed.
         Your password has been given incorrectly once since last correct use.
         You are protected from preemption.
         u1 p1 logged in 01/01/75  1200.0 edt Mon from ASCII terminal "102".
         Last login 01/01/75  1100.0 edt Fri from ASCII terminal "101".
         r 1201 3.271 1.010 35

        *create mailbox
         r 1201 1.023 .023 12

        *list

         Segments = 1, Records = 0.

         r w    0  mailbox

         r 1202 .034 .120 15

        *logout
         u1 p1 logged out 01/01/75  1203.2 edt Mon
         CPU usage 13 sec, memory usage 13.5 units.
        hangup

PDS-3:  (CONTINUE)
        (User puts terminal online)
        Multics 1.1.1: AF Data Services Center.
        Load = 27.0 out of 50.0 units: users = 27
       *login u1
        Password:
       *pass1
        Login incorrect.
        Please try again or type "help" for instructions.
       *login u1
        Password:
       *arptoa
        You are protected from preemption.
        Your password has been given incorrectly once since last correct use.
        u1 p1 logged in 01/01/75  1213.4 edt Mon from ASCII terminal "102".
        Last login 01/01/75  1200.0 edt Mon from ASCII terminal "102".
        r 1213 3.125 12.023 152

       *logout
        u1 p1 logged out 01/01/75  1215.5 edt Mon
        CPU usage 7 sec, memory usage 152.4 units.
        hangup

PDS-4:  (CONTINUE)
        (user puts terminal online)
       *login u1 -generate_password
        Password:
       *arptoa
        Your new password is "bakops", pronounced "ba-kops".
       *Please type the password: nibno
        Password changed.
        You are protected from preemption.
        u1 p1 logged in 01/01/75  1215.7 edt from ASCII terminal "102".
        Last login 01/01/75  1213.4 edt Mon from ASCII terminal "102".
        r 1217 3.253 15.434 122

       *listacl mailbox

        r w  u1.p1.*
        rew  *.SysDaemon.*

        r 1217 .201 1.023 12
```

```
         *logout
          u1 p1 logged out 01/01/75  1219.2 edt Mon
          CPU usage 7 sec, memory usage 123.8 units.
          hangup

PDS-5:   (CONTINUE)
          (user puts terminal online)
          Multics 1.1.1: AF Data Services Center.
          Load = 25 out of 50.0 units: users = 25
         *login u1
          Password:
         *bakops
          Login incorrect.
          Please try again or type "help" for instructions.
         *login u1
          Password:
         *nibno
          Your password has been given incorrectly once since last correct use.
          You are protected from preemption.
          u1 p1 logged in 01/01/75  1222.4 edt Mon from ASCII terminal "102".
          Last login 01/01/75  1219.2 edt Mon from ASCII terminal "102".
          r 1220 3.333 14.234 199

         *logout
          u1 p1 logged out 01/01/75  1221.2 edt Mon
          CPU usage 4 sec, memory usage 15.3 units.
          hangup

PDS-6:   (user puts terminal online)
          Multics 1.1.1: AF Data Services Center.
          Load = 27.0 out of 50.0 units: users = 27
         *login u2 -generate_password
          Password:
         *xxxxxx (This should be u2's current password)
          Your new password is "rofine", pronounced "ro-fine".
         *Please type the password: xxxxxx (Same password as above)
          Incorrect.
          Your new password is "grece", pronounced "grece".
         *Please type the new password: grece
          Password changed.
          You are protected from preemption.
          u2 p1 logged in 01/01/75  1230.2 edt Mon from ASCII terminal "102".
          Last login 01/01/75  1222.4 edt Mon from ASCII terminal "102".
          r 1231 3.222 4.542 12
```

```
        *logout
         u2 p1 logged out 01/01/75  1232.0 edt Mon
         CPU usage 4 sec, memory usage 14.2 units.
         hangup

PDS-7:   (CONTINUE)
         (user puts terminal online)
         Multics 1.1.1: AF Data Services Center.
         Load = 27.0 out of 50.0 units: users = 27
        *login u2 -change_password
         Password:
        *grece
         Login incorrect.
         You must use the -generate_password option to change your password.
         Please try again or type "help" for instructions.
         (user hangs up)
```

(The last password assigned to u2 in PDS-6 (and used in PDS-7) should be remembered for subsequent logins. Alternatively, the SA can login at this point and change the password of u2 back to what it was originally.)

PROCESS AUTHORIZATION ASSIGNMENT (PAA)

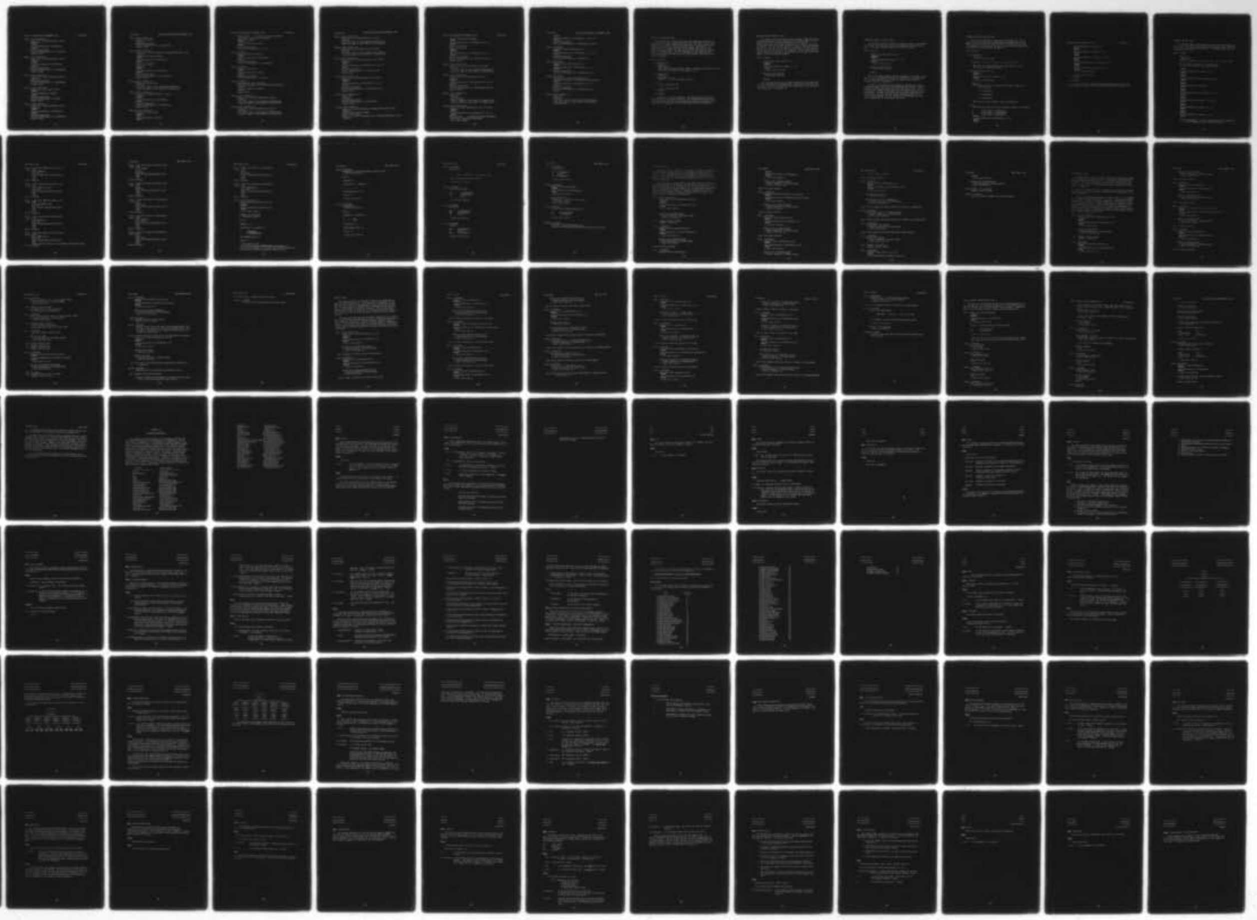
The scripts of the process authorization tests described in Section III (beginning at page 38) are presented below. The scripts are numbered PAA-1 through PAA-34, corresponding to the test numbers in the text of Section III. The users and projects referred to in these scripts are those set up in the test environment discussed in Section III (on page 26) and detailed in Appendix I.

For conciseness, the -brief option is always specified in the login line and on logouts. It is not necessary to specify this option when actually running the tests, although it is recommended that the option be used at least once on a legal login to check that the printing of the process authorization is not suppressed. Ready message and password input have been omitted from the scripts.

```
PAA-1:  (from terminal t1)
       *login u1 p1 -auth secret
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.

PAA-2:  (from terminal t1)
       *login u3 p1 -auth confidential
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.

PAA-3:  (from terminal t1)
       *login u2 p1 -auth secret
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.

PAA-4:  (from terminal t2)
       *login u1 p1
        Password:
        Login incorrect.
        Please try again or type "help" for instructions.
        (Computer operator is notified of illegal login.)

PAA-5:  (from terminal t2)
       *login u2 p1 -auth top_secret
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.
```

AD-A034 986 MITRE CORP BEDFORD MASS F/G 15/3
TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS,(U)
DEC 76 M GASSER, S R AMES, L J CHMURA F19628-76-C-0001
UNCLASSIFIED MTR-3005 ESD-TR-76-164 NL

2 OF 4
ADA034986



```
PAA-6:  (from terminal t1)
        *login u1 p1 -auth confidential -brief
         Password:
         ******
         Your authorization is confidential.
         ******
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief

PAA-7:  (from terminal t1)
        *login u2 p1 -auth confidential -brief
         Password:
         ******
         Your authorization is confidential.
         ******
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief

PAA-8:  (from terminal t1)
        *login u3 p1 -auth unclassified -brief
         Password:
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief

PAA-9:  (from terminal t2)
        *login u2 p1 -auth secret -brief
         ******
         Your authorization is secret.
         ******
      > *authorization_tester
         Process authorization is: secret.
        *logout -brief

PAA-10: (from terminal t1)
        *login u2 p1 -auth confidential -brief
         Password:
         ******
         Your authorization is confidential.
         ******
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief
```

```
PAA-11: (from terminal t2)
        *login u1 p1 -brief
         Password:
      > *authorization_tester
         Process authorization is: unclassified.
        *logout -brief

PAA-12: (from terminal t2)
        *login u2 p1 -auth confidential -change_default_auth -brief
         Password:
         Default authorization changed.
         ******
         Your authorization is confidential.
         ******
        *logout -brief -hold
        *login u2 p1 -brief
         Password:
         ******
         Your authorization is confidential.
         ******
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief

PAA-13: (from terminal t2)
        *login u2 p2 -auth secret
         Password:
         You cannot login at the requested authorization.
         Please try again or type "help" for instructions.

PAA-14: (from terminal t2)
        *login u2 p2 -auth confidential -brief
         Password:
         ******
         Your authorization is confidential.
         ******
      > *authorization_tester
         Process authorization is: confidential.
        *logout -brief

PAA-15: (from terminal t2)
        *login u2 p1 -auth secret
         Password:
         ******
         Your authorization is secret.
         ******
```

```
       *abnormal_term      (terminates process abnormally)
        Fatal error. Process has terminated.
        New process created.
        ******
        Your authorization is secret.
        ******
     > *authorization_tester
        Process authorization is: secret.

PAA-16: (CONTINUE)
       *new_proc -auth confidential
        ******
        Your authorization is confidential.
        ******
     > *authorization_tester
        Process authorization is: confidential.

PAA-17: (CONTINUE)
       *new_proc -auth secret
        ******
        Your authorization is secret.
        ******
     > *authorization_tester
        Process authorization is: secret.

PAA-18: (CONTINUE)
     > *new_proc_test -auth top_secret
        You cannot new_proc to the requested authorization.
        ******
        Your authorization is secret.
        ******
       *logout -brief

PAA-19: (from terminal t3)
       *login u4 p3 -auth confidential,c1,c2,c6,c7
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.

PAA-20: (from terminal t3)
       *login u4 p3 -auth confidential,c1,c3,c6,c7
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.
```

```
PAA-21: (from terminal t3)
        *login u4 p3 -auth confidential,c1,c4,c6,c7
         Password:
         You cannot login at the requested authorization.
         Please try again or type "help" for instructions.

PAA-22: (from terminal t3)
        *login u4 p3 -auth confidential,c1,c5,c6,c7
         Password:
         You cannot login at the requested authorization.
         Please try again or type "help" for instructions.

PAA-23: (from terminal t3)
        *login u4 p3 -auth confidential,c1,c6,c7 -brief
         Password:
         ******
         Your authorization is confidential, c1, c6, c7.
         ******
      > *authorization_tester
         Process authorization is: secret,c1,c6,c7.
        *logout -brief

PAA-24: (from terminal t3)
        *login u4 p3 -auth confidential,c1,c6 -brief
         Password:
         ******
         Your authorization is confidential, c1, c6.
         ******
      > *authorization_tester
         Process authorization is: confidential,c1,c6.
        *logout -brief

PAA-25: (from terminal t3)
        *login u4 p3
         Password:
      > *authorization_tester
         Process authorization is: unclassified.
        *logout -brief

PAA-26: (from terminal t3)
        *login u5 p3 -auth unclassified -change_default_auth -brief
         Password:
         Default authorization changed.
        *logout -hold -brief
        *login u5 p3 -auth confidential,c6,c7 -change_default_auth -brief
         Password:
```

```
        Default authorization changed.
        ******
        Your authorization is confidential, c6, c7.
        ******
       *logout -brief -hold
       *login u5 p3 -brief
        Password:
        ******
        Your authorization is confidential, c6, c7.
        ******
     > *authorization_tester
        Process authorization is: confidential,c6,c7.
       *logout -brief

PAA-27: (from terminal t3)
       *login u4 p4 -auth confidential,c4,c5,c6,c7
        Password:
        You cannot login at the requested authorization.
        Please try again or type "help" for instructions.

PAA-28: (from terminal t3)
       *login u4 p4 -auth confidential,c4,c5,c6
        Password:
        ******
        Your authorization is confidential, c4, c5, c6.
        ******
     > *authorization_tester
        Process authorization is: confidential,c4,c5,c6.
       *logout -brief

PAA-29: (from terminal t4)
       *login u4 p3
        Password:
        Login incorrect.
        Please try again or type "help" for instructions.
        (Computer operator is notified of illegal login.)

PAA-30: (from terminal t3)
       *login u4 p3 -auth confidential,c1,c6,c7 -brief
        Password:
        ******
        Your authorization is confidential, c1, c6, c7.
        ******
       *abnormal_term      (terminates process abnormally)
        Fatal error. Process has terminated.
        New process created.
        ******
        Your authorization is confidential, c1, c6, c7.
        ******
     > *authorization_tester
        Process authorization is: confidential,c1,c6,c7.

PAA-31: (CONTINUE)
       *new_proc -auth confidential,c1,c6
        ******
        Your authorization is confidential, c1, c6, c7.
        ******
     > *authorization_tester
        Process authorization is: confidential.

PAA-32: (CONTINUE)
       *new_proc -auth confidential,c1,c7
        ******
        Your authorization is confidential, c1, c7.
        ******
     > *authorization_tester
        Process authorization is: confidential,c1,c7.

PAA-33: (CONTINUE)
     > *new_proc_test -auth confidential,c1,c4
        You cannot new_proc to the requested authorization.
        ******
        Your authorization is confidential, c1, c7.
        ******
     > *authorization_tester
        Process authorization is: confidential,c1,c7
       *logout

PAA-34: (from terminal t3)
       *login u5 p4
        Password:
        Your cannot login at your default authorization.
        Please try again or type "help" for instructions.
        (user hangs up)
```

SEGMENT ACL CONTROLS (SAC)

The segment ACL controls tests described beginning on page 45 are performed by a single command executed by the user while at some fixed authorization, either system_low or otherwise. The tests must be performed by two processes having different process identifiers. This means that the two processes must have different user names and/or different project names. The authorizations of the users are unimportant -- unclassified can be used. Although any user names and projects may be used, the users u1 and u2 and projects p1 and p2 are used below as examples.

SAC-1 to SAC-25:

```
 u1: *login u1 p1
      Password:
   > *test_seg_acl
      *** Login at second terminal, under a different name and/or project.
      *** Issue the command: "test_seg_acl u1 p1"

 u2: *login u2 p2
      Password:
   > *test_seg_acl u1 p1
      *** Return to first terminal, type an s.

 u1: *s
      r 959 4.440 56.069 493

 u2:  r 575 4.233 43.233 123
     *logout

 u1: *logout
```

Appearance of a ready message on both terminals indicates proper completion of the test with no errors. The correspondence between the test numbers and the code that implements the test can be found in the listings of test_seg_acl.pl1 and supporting routines in Appendix IV.

SEGMENT SECURITY CONTROLS (SSC)

The segment security controls tests discussed on page 52 are performed by executing a single command from a process logged in at an authorization equal to the third argument in the call to create_test_seg.ec during test initialization as shown at the middle of page 86. Any user with the ability to login at that authorization may be used. In this script, user u7 on project p5 is used, with authorization secret,c1,c2 as in the examples. The directory SEG_TEST, created during initialization, is referenced in the script as the argument to the test_seg_auth command.

SSC-1 to SSC-12:

```
    *login u7 p5 -auth secret,c1,c2
     Password:
     ******
     Your authorization is secret, c1, c2.
     ******
     r 1230 4.344 23.544 576

  > *test_seg_auth SEG_TEST
     r 1231 0.608 3.234 455

    *logout
```

The ready message indicates proper termination of the test with no errors detected. The correspondence between the test number and the actual code that performs the test can be found in the listing of test_dir_auth.pl1 in Appendix IV.

DIRECTORY SECURITY CONTROLS (DSC)

The directory security controls, discussed on page 54, are tested by a script very similar to that for the segment security controls (SSC). That same discussion applies to the script below.

DSC-1 to DSC-17:

```
     *login u7 p5 -auth secret,c1,c2
      Password:
      ******
      Your authorization is secret, c1, c2.
      ******
      r 1230 4.444 2.343 122

   > *test_dir_auth DIR_TEST
      r 1231 15.222 4.328 154

     *logout
```

Again, the ready message indicates completion of the test with no errors. The correspondence between the individual test numbers and the code that performs the test can be found in the listing of test_dir_auth.pl1 in Appendix IV.

Failure of the test_dir_auth command as indicated by an error message may simply be due to a system change that returns a slightly different status code in some call to an hcs_ entry point. The test_dir_auth command usually terminates the test when any error is encountered. In order to determine all the status codes that have been changed, the user may call check_status_$return before the test_dir_auth command. See the writeup of check_status_ for a detailed explanation of the operation involved.

INTERPROCESS COMMUNICATION (IPC)

The IPC tests discussed on page 57 are performed by two processes, which may or may not be originated by the same user. In the example in the script below, users u6 and u7 on project p5 are used. It is assumed that user u6 has the call to test_ipc in his start_up.ec as described in initialization on page 85.

IPC-1 to IPC-6:

```
u6: *login u6 p5
     Password:
     r 1202 5.143 35.221 456

   > *test_ipc  C,1,2  S,1  S,1,2  S,1,2,3  T,1,2  S,1,3

     *** Login at second terminal with authorization = S,1,2
     *** Issue the command "test_ipc".

u7: *login u7 p5 -auth S,1,2
     Password:
     ******
     Your authorization is secret, c1, c2.
     ******
     r 1205 5.133 35.344 567

   > *test_ipc
     *** You will need the following three numbers, back at the
          first terminal.

          1.) 012345673416

          2.) 750364263444

          3.) 463764263035

     *** Return to first terminal, type the character s.

u6: *s
     *** Using the output from other terminal, answer the following.

    *          First number = 012345673416
    *          Second number = 750364263444
    *          Third number = 463764263035
     ******
     Your authorization is confidential, c1, c2.
     ******
     ******
```

```
        Your authorization is secret, c1.
        ******
        ******
        Your authorization is secret, c1, c2.
        ******
        ******
        Your authorization is secret, c1, c2, c3.
        ******
        ******
        Your authorization is top secret, c1, c2.
        ******
        ******
        Your authorization is secret, c1, c3.
        ******
        r 1206 5.322 34.450 568

   u7:  r 1206 2.343 34.232 456
       *logout

   u6: *logout
```

The source code that implements the individual tests can be found in the listing of test_ipc.pl1 and supporting routines in Appendix IV.

MESSAGE SEGMENTS (MBX)

The message segment tests discussed on page 60 are performed by a single user. Any user on any terminal able to login to at least three levels and three categories may be used. For these scripts, user u7 on project p5 is used.

MBX-1 to MBX-6:

```
      *login u7 p5
       Password:
    > *exec_com mbx_test  C,1,2  T,1,2  S,1,2  S,1  S,1,2,3  S,1,3

      *Do you want to delete your old mailbox? Yes.
       Please ignore the next six "Input:" lines.

       ******
       Your authorization is secret, c1, c2.
       ******
       Input:
       ******
       Your authorization is confidential, c1, c2.
       ******
       Input:
       ******
       Your authorization is secret, c1.
       ******
       Input:
       ******
       Your authorization is top secret, c1, c2.
       ******
       Input:
       ******
       Your authorization is secret, c1, c2, c3.
       ******
       Input:
       ******
       Your authorization is secret, c1, c3.
       ******
       Input:
       ******
       Your authorization is secret, c1, c2.
       ******

       mbx_test: Messages A, B, and C should follow, plus "Incorrect
             access" messages from mail regarding 2 and 3:
```

```
3 messages.

1) From: u7.p5  01/01/75  1200.0 edt Mon (1 line)

A. This message is S,1,2.

2) From: u7.p5  01/01/75  1200.5 edt Mon (1 line)

B. This message is C,1,2.

3) From: u7.p5  01/01/75  1200.9 edt Mon (1 line)

C. This message is S,1.

mail: Incorrect access on entry. Message 2 not deleted.
mail: Incorrect access on entry. Message 3 not deleted.

mbx_test: Messages B and C should now follow:

2 messages.

1) From: u7.p5  01/01/75  1200.5 edt Mon (1 line)

B. This message is C,1,2.

2) From: u7.p5  01/01/75  1200.9 edt Mon (1 line)

C. This message is S,1.

mbx_test: One final error message from mbx_delete:

mbx_delete: Incorrect access to directory containing entry.
      >udd>p5>u7>u7.mbx

*mbx_test: Everything as expected? yes
r 1210 3.456 12.324 456

*logout
```

The commands that implement the tests can be found in the listing of aim_test_exec_coms in Appendix IV. There is no direct correspondence between test number and lines of code, however, because the test numbers merely refer to the individual messages that are sent or the various authorizations. See the writeup of mbx_test.ec in Appendix III for a description of the six arguments to mbx_test.

CARD INPUT (CIF)

The scripts of the card deck input tests described in Section III (beginning on page 62) are presented below. The scripts are numbered CIF-1 through CIF-24, corresponding to the test numbers in the text of Section III. The user and projects referred to in these scripts are those set up in the test environment discussed in Section III (on page 26) and detailed in Appendix I.

For each of the tests CIF-1 to CIF-14 the card reader is to be logged in at the indicated authorization. These authorizations have been set up at initialization as shown in the table on page 89. The first 14 tests consist of reading cards into the card reader as indicated, where each line in the script represents one card. The symbols <UID> and <EOF> represent a unique id and end of file card respectively. The result, where shown, indicates whether the cards were accepted or rejected. A reject does not necessarily mean that the card reader refuses to read to the end of the deck. However, the operator should be notified of the error.

Although the first 14 tests are independent, it is not necessary to login the card reader for each test if the card reader did not log itself out after the previous test. The only requirement is that the authorization of the card reader is as specified.

```
CIF-1:  (login card reader at secret.c1,c2)
 cards: <UID>
        U7.P5 SECRET,C1,C2;
        CIF-1 MCC
        "THIS IS TEST DATA FOR TEST CIF-1"
        <EOF>
        <UID>
result: accepted

CIF-2:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5 UNCLASSIFIED,C1,C2;
        CIF-2 MCC
        "THIS IS TEST DATA FOR TEST CIF-2"
        <EOF>
        <UID>
result: rejected
```

```
CIF-3:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5 SECRET,C1;
        CIF-3 MCC
        "THIS IS TEST DATA FOR TEST CIF-3"
        <EOF>
        <UID>
result: rejected

CIF-4:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5 SECRET,C1,C2,C3;
        CIF-4 MCC
        "THIS IS TEST DATA FOR TEST CIF-4"
        <EOF>
        <UID>
result: rejected

CIF-5:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5 TOP SECRET,C1,C2;
        CIF-5 MCC
        "THIS IS TEST DATA FOR TEST CIF-5"
        <EOF>
        <UID>
result: rejected

CIF-6:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5 SECRET,C3;
        CIF-6 MCC
        "THIS IS TEST DATA FOR TEST CIF-6"
        <EOF>
        <UID>
result: rejected

CIF-7:  (login card reader at secret,c1,c2)
 cards: <UID1>
        U6.P5 SECRET,C1,C2;
        CIF-7 MCC
        "THIS IS TEST DATA FOR TEST CIF-7"
        <EOF>
        <UID2> (this UID card is to be different from <UID1> above)
result: rejected
```

```
CIF-8:  (login card reader at secret,c1,c2)
 cards: <UID>
        U7.P5 SECRET,
        C1,C2;
        CIF-8 mcc
        "THIS IS TEST DATA FOR TEST CIF-8"
        <EOF>
        <UID>
result: accepted

CIF-9:  (login card reader at secret,c1,c2)
 cards: <UID>
        U6.P5;
        CIF-9 MCC
        "THIS IS TEST DATA FOR TEST CIF-9"
        <EOF>
        <UID>
result: rejected

CIF-10:  (login card reader at unclassified)
 cards: <UID>
        U7.P5;
        CIF-10 MCC
        "THIS IS TEST DATA FOR TEST CIF-10"
        <EOF>
        <UID>
result: accepted

CIF-11: (login card reader at unclassified)
 cards: <UID>
        U6.P5 UNSECRET;
        CIF-11 MCC
        "THIS IS TEST DATA FOR TEST CIF-11"
        <EOF>
        <UID>
result: rejected

CIF-12: (login card reader at unclassified)
 cards: <UID>
        *.P5;
        CIF-12 MCC
        "THIS IS TEST DATA FOR TEST CIF-12"
        <EOF>
        <UID>
result: rejected
```

```
CIF-13: (login card reader at unclassified)
 cards: <UID>
        U6.P5;
        CIF-13 qqq
        "THIS IS TEST DATA FOR TEST CIF-13"
        <EOF>
        <UID>
result: rejected

CIF-14: (login card reader at unclassified)
 cards: <UID>
        U7.P5 UNCLASSIFIED;
        CIF-10 MCC
        "THIS IS TEST DATA FOR TEST CIF-14"
        <EOF>
        <UID>
result: accepted

CIF-15: (from terminal t5)
       *login u7 p5 -auth secret,c1,c2 -bf
        Password:
        ******
        Your authorization is secret,c1,c2.
        ******
        r 1737 0.362 1.482 37

       *change_wdir >ddd>cards
        r 1738 0.055 0.546 24

       *list -a

        Segments = 0.

        Directories = 2, Records = 2.

        s     1 system_low
        s     1 !bBBBBBBBBBJBBB

        Multisegment-files = 0.

        Links = 0.

        r 1738 0.649 2.252 45
        (The directory named !bBBBBBBBBBJBBB corresponds to a
        secret,c1,c2 directory.  It is the value returned from the ac-
        tive function [encode_authorization secret,c1,c2].)
```

```
CIF-16: (CONTINUE)
        *change_wdir [encode_authorization secret,c1,c2]
         r 1738 0.065 0.546 24

        *list -a

         Segments = 0.

         Directories = 1, Records = 1.

         s     1     u7

         Multisegment-files = 0.

         Links = 0.

         r 1738 0.183 0.342 9


CIF-17: (CONTINUE)
        *change_wdir u7
         r 1738 0.060 0.672 38

        *list -a

         Segments = 2  Records = 2.

         r     1    cif-1
         r     1    cif-8

         Directories = 0.

         Multisegment-files = 0

         Links = 0.

         r 1739 0.156 1.700 31
```

CIF-13: (CONTINUE)

```
        *print cif-1


                cif-1    06/21/75    1752.3  edt  Sat

        "this is test data for test cif-1"

        r 1752 0.267 1.122 32
```

CIF-19: (CONTINUE)

```
        *list_acl >ddd>cards

        sma          IO.SysDaemon.*
        sma          *.SysDaemon.*
        s            *.*.*

        r 1759 0.160 4.686 82
```

CIF-20: (CONTINUE)

```
        *list_acl <

        sma       IO.SysDaemon.*
        sma       *.SysDaemon.*
        s         *.*.*

        r 1800 0.116 0.306 16
```

CIF-21: (CONTINUE)

```
        *list_acl

        sma       IO.SysDaemon.*
        s         u7.*.*
        sma       *.SysDaemon.*

        r 1801 0.134 3 194 56
```

```
CIF-22: (CONTINUE)
        *list_acl cif-1

         rw    IO.SysDaemon.*
         r     u7.p5.*
         rw    *.SysDaemon.*

         r 1802 0.391 6.714 94


CIF-23: (CONTINUE)
        *new_proc -auth unclassified
         ******
         Your authorization is unclassified
         *****
         r 1802 0.924 8.930 88

        *copy_cards cif-10
         copy_cards:  2 copies of cif-10 exist
         1 card decks copied
         r 1803 0.061 0.554 27

CIF-24: (CONTINUE)
        *list_acl -pn >system_library_1>ioi_

         re        IO.SysDaemon.*
         re        *.SysDaemon.*

         r 1803 0.053 0.645 29


OPERATOR: (CLEANUP)
        delete_dir >udd>cards>system_low>u7
        delete_dir >udd>cards>[encode_authorization secret,c1,c2]>u7
```

CARD OUTPUT (CPT)

The scripts of the card punch described in Section III (beginning on page 66) are presented below. The scripts are numbered CPT-1 through CPT-11 corresponding to the test numbers in the text of Section III. The users and projects referred to in these scripts are those set up in the test environment discussed in Section III and detailed in Appendix I.

The punch must be initialized to the access class and other parameters specified in the table on page 89 during initialization. It is assumed that the punch is turned on and that the queue is initially empty. Following each dpunch command the I/O coordinator returns information regarding the number of requests signalled by the dpunch command and the number of requests already queued for output. These numbers should be checked for correctness.

```
CPT-1:  (from terminal t5)
       *login u7.p5 -auth confidential,c1,c2 -bf
        Password:
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1551 0.760 2.498 47


       *dpunch -mcc IO_TEST>OUTPUT
        1 request signalled, 0 already queued
        r 1553 0.719 12.098 149

 punch: Segment punched: OUTPUT
        Banner:  R,c1,c2,c3

CPT-2:  (CONTINUE)
       *new_proc -auth unclassified,c1,c2
        ******
        Your authorization is unclassified,c1,c2.
        ******
        r 1554 0.941 9.554 119

       *dpunch -mcc IO_TEST>NO_OUTPUT
        1 request signalled, 0 already queued
        r 1555 0.258 0.420 23

 punch: No output punched.

CPT-3:  (CONTINUE)
       *new_proc -auth confidential,c1
```

```
        ******
        Your authorization is confidential,c1.
        ******
        r 1556 0.920 6.806 78

       *dpunch -mcc IO_TEST>NO_OUTPUT
        1 request signalled, 0 already queued
        r 1556 1.817 16.704 123

 punch: No output punched.

CPT-4:  (CONTINUE)
       *new_proc -auth confidential,c1,c2,c3,c4,c5
        ******
        Your authorization is confidential,c1,c2,c3,c4,c5.
        ******
        r 1557 0.920 6.806 78

       *dpunch -mcc IO_TEST>NO_OUTPUT
        1 request signalled, 2 already queued.
        r 1558 0.218 1.332 33

 punch: No output punched.

CPT-5:  (CONTINUE)
       *new_proc -auth top_secret,c1,c2
        ******
        Your authorization is top_secret,c1,c2.
        ******
        r 1559 0.921 6.806 78

       *dpunch -mcc IO_TEST>NO_OUTPUT
        1 request signalled, 2 already queued
        r 1560 1.774 16.704 123

 punch: No output produced.

CPT-6:  (CONTINUE)
       *new_proc -auth confidential,c2,c3
        ******
        Your authorization is confidential,c2,c3.
        ******
        r 1560 0.920 6.806 78

       *dpunch -mcc IO_TEST>NO_OUTPUT
        1 request signalled, 0 already queued.
        r 1561 1.803 17.616 181

 punch: No output produced

CPT-7:  (CONTINUE)
       *new_proc -auth confidential,c1,c2
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1562 0.921 7.482 81

       *change_wdir IO_TEST
        r 1563 0.060 0.672 12

     > *dpunch_test -mcc S,1,2>BAD_LEVEL
        1 request signalled, 2 already queued.
        r 1563 1.817 16.704 153

 punch: Error message indicating inability to access S,1,2>BAD_LEVEL

CPT-8:  (CONTINUE)
     > *dpunch_test -mcc C,1,2,3>BAD_CATEGORY
        1 request signalled, 2 already queued.
        r 1564 1.817 28.020 163

 punch: Error message indicating inability to access C,1,2,3>BAD_CATEGORY

CPT-9:  (CONTINUE)
     > *dpunch_test -mcc BAD_ACL
        1 request signalled 2 already queued.
        r 1564 0.233 2.678 54

 punch: Error message indicating inability to access BAD_ACL

CPT-10: (CONTINUE)
       *dpunch -mcc OUTPUT
        1 request signalled 2 already queued
        r 1565 1.830 17.616 181

 punch: Banner:  R,c1,c2,c3
        Segment punched: OUTPUT

CPT-11: (CONTINUE)
       *new_proc -auth secret,c1,c2,c3,c4
        ******
        Your authorization is secret,c1,c2,c3,c4.
        ******
        r 1566 0.920 6.806 78

       *dpunch -mcc IO_TEST>OUTPUT
        1 request signalled 3 already queued.
        r 1567 0.86 3.948 56

 punch: Banner:  S,c1,c2,c3,c4
        Segment punched: OUTPUT

OPERATOR: (CLEANUP)
        Delete all dpunch requests from the punch queue.
```

LINE PRINTER (LPT)

The scripts for the line printer described in Section III (beginning on page 68) are presented below. The scripts are numbered LPT-1 through LPT-22 corresponding to the test numbers in the text of Section III. The users and projects referred to in these scripts are those set up in the test environment discussed in Section III and detailed in Appendix I.

The access classes for the two printers prt1 and prt2, and other attributes, are specified during initialization as summarized in the table on page 89.

Initially prt1 should be logged in, but not prt2. It is assumed that the dprint queues are empty at the start. Following each dprint command the I/O coordinator returns information regarding the number of requests signalled by the dprint command and the number of requests already queued for output. These numbers should be checked for correctness. They will be correct only if the printer has completed printing the results of the previous requests. Thus, the user should wait for the printer to complete any output from the previous test before continuing with the next test.

```
LPT-1:  (from terminal t5)
       *login u7 p5 -auth confidential,c1,c2 -bf
        Password:
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1551 0.760 2.498 47

       *dprint IO_TEST>OUTPUT
        1 request signalled, 0 already queued
        r 1553 0.719 12.098 149

  prt1: Segment printed: OUTPUT
        Banner:  R,c1,c2,c3

LPT-2:  (CONTINUE)
       *new_proc -auth unclassified,c1,c2
        ******
        Your authorization is unclassified,c1,c2.
        ******
        r 1554 0.941 9.554 119
        *dprint IO_TEST>NO_OUTPUT
         1 request signalled, 0 already queued
         r 1555 0.258 0.420 23

  prt1: No output printed.

LPT-3:  (CONTINUE)
        *new_proc -auth confidential,c1
         ******
         Your authorization is confidential,c1.
         ******
         r 1556 0.920 6.806 78

        *dprint IO_TEST>NO_OUTPUT
         1 request signalled, 0 already queued
         r 1556 1.817 16.704 123

  prt1: No output printed.

LPT-4:  (CONTINUE)
        *new_proc -auth confidential,c1,c2,c3,c4,c5
         ******
         Your autnorization is confidential,c1,c2,c3,c4,c5.
         ******
         r 1557 0.920 6.806 78

        *dprint IO_TEST>NO_OUTPUT
         1 request signalled, 2 already queued.
         r 1558 0.218 1.332 33

  prt1: No output printed.

LPT-5:  (CONTINUE)
        *new_proc -auth top_secret,c1,c2
         ******
         Your authorization is top_secret,c1,c2.
         ******
         r 1559 0.921 6.806 78

        *dprint IO_TEST>NO_OUTPUT
         1 request signalled, 2 already queued
         r 1560 1.774 16.704 123

  prt1: No output produced.
```

```
LPT-6:  (CONTINUE)
       *new_proc -auth confidential,c2,c3
        ******
        Your authorization is confidential,c2,c3.
        ******
        r 1560 0.920 6.806 78

       *dprint IO_TEST>NO_OUTPUT
        1 request signalled, 0 already queued.
        r 1561 1.803 17.616 181

  prt1: No output produced

LPT-7:  (CONTINUE)
       *new_proc -auth confidential,c1,c2
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1562 0.921 7.482 81

       *change_wdir IO_TEST
        r 1563 0.060 0.672 12

     > *dprint_test S,1,2>BAD_LEVEL
        1 request signalled, 2 already queued.
        r 1563 1.817 16.704 153

  prt1: Error message indicating inability to access S,1,2>BAD_LEVEL

LPT-8:  (CONTINUE)
     > *dprint_test C,1,2,3>BAD_CATEGORY
        1 request signalled, 2 already queued.
        r 1564 1.817 28.020 163

  prt1: Error message indicating inability to access C,1,2,3>BAD_CATEGORY

LPT-9:  (CONTINUE)
     > *dprint_test BAD_ACL
        1 request signalled, 2 already queued.
        r 1564 0.233 2.678 54

  prt1: Error message indicating inability to access BAD_ACL

LPT-10: (CONTINUE)
       *dprint OUTPUT
        1 request signalled, 2 already queued
        r 1565 1.830 17.616 181

  prt1: Segment printed: OUTPUT
        Banner: R,c1,c2,c3

LPT-11 & LPT-12: (CONTINUE)
       *new_proc -auth secret,c1,c2,c3,c4
        ******
        Your authorization is secret,c1,c2,c3,c4.
        ******
        r 1566 0.920 6.806 78

       *change_wdir IO_TEST
        r 1566 0.060 0.672 12

       *dprint OUTPUT
        1 request signalled, 3 already queued
        r 1615 0.922 5.512 78

  prt1: Segment printed: OUTPUT
        Banner: S,c1,c2,c3,c4
        Top page label: unclassified
        Bottom page label: unclassified

LPT-13: (CONTINUE)
       *dprint -access_label OUTPUT
        1 request signalled, 3 already queued
        r 1616 2.033 33.206 187

  prt1: Segment printed: OUTPUT
        Top page label: unclassified
        Bottom page label: unclassified

LPT-14: (CONTINUE)
       *dprint -label "This is confidential" OUTPUT
        1 request signalled, 3 already queued.
        r 1626 0.435 13.116 109

  prt1: Segment printed: OUTPUT
        Top page label: This is confidential
        Bottom page label: This is confidential
```

```
LPT-15: (CONTINUE)
       *dprint -top_label "This is a top label" OUTPUT
        1 request signalled, 3 already queued.
        r 1632 0.243 4.314 71

  prt1: Segment printed: OUTPUT
        Top page label: This is a top label
        Bottom page label: unclassified

LPT-16: (CONTINUE)
       *dprint -bottom_label "This is a bottom label" OUTPUT
        1 request signalled, 3 already queued
        r 1637 0.251 4.678 68

  prt1: Segment printed: OUTPUT
        Top page label: unclassified
        Bottom page label: This is a bottom label

LPT-17: (CONTINUE)
        (Operator brings up printer prt2)

       *dprint LONG LONG
        2 requests signalled, 3 already queued
        r 1638 1.589 18.754 216

  prt1: Segment printed: LONG
        Banner: S,c1,c2,c3
  prt2: Segment printed: LONG
        Banner: T,c1,c2,c3,c4

LPT-18: (CONTINUE)
       *new_proc -auth top_secret,c1,c2,c3,c4
        ******
        Your authorization is top secret,c1,c2,c3,c4.
        ******
        r 1639 0.930 9.120 92

       *dprint IO_TEST>LONG IO_TEST>LONG
        2 requests signalled, 4 already queued
        r 1640 0.938 3.102 80

  prt1: No output.
  prt2: Segments printed: LONG (2 copies)
        Banners: TS,c1,c2,c3,c4
```

```
LPT-19: (CONTINUE)
       *new_proc -auth secret,c1,c2,c3,c4,c5
        ******
        Your authorization is secret,c1,c2,c3,c4,c5.
        ******
        r 1640 0.941 9.554 119

       *dprint IO_TEST>LONG IO_TEST>LONG
        2 requests signalled, 4 already queued
        r 1641 2.177 22,112 144

  prt1: No output
  prt2: Segments printed: LONG (2 copies)
        Banners: T,c1,c2,c3,c4,c5

LPT-20: (CONTINUE)
        (Operator should check that there is one accountability form
         for each of the tests LPT-1, and LPT-7 through LPT-19.  The
         information appearing on the accountability form should be
         checked for correctness.)

LPT-21: (Operator brings up prt1 but not the accountability terminal)
       *login u7 p5 -auth confidential,c1,c2 -bf
        Password:
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1642 0.362 1.482 37

       *change_wdir IO_TEST
        r 1643 0.065 0.546 24

       *dprint LONG LONG
        2 requests signalled, 2 already queued
        r 1643 0.921 8.190 86

   prt1: No output is produced because accountability terminal not
            dialed up.

LPT-22: (CONTINUE)
        (Operator dials up accountability terminal for prt1)

  prt1: Segment LONG begins printing.

        (Operator disconnects accountability terminal for prt1 while
            prt1 is still printing first copy of LONG.)
```

```
prt1: Second copy of segment LONG not printed.

OPERATOR: (CLEANUP)
          Delete all dprint requests from the print queue.
```

TAPE I/O (TDT)

The tests for tape I/O described in Section III (beginning on page 72) are presented below. The tape drive is assumed to be initialized to the values of the parameters shown in the table on page 89. For these tests the operator will be requested by the system to mount a tape having the identifier "reel_id". This single tape required for the tape I/O tests can be any scratch tape. An attempt will be made to read and write the tape at several different access classes.

Following each read_tape and write_tape command the I/O coordinator returns information regarding the number of requests signalled by the command and the number of requests already queued for input or output. It is assumed that there are separate queues for reading and writing tapes and that these queues are initially empty. With these assumptions the numbers supplied in the scripts are correct and should be checked.

```
TDT-1:  (from terminal t5)
       *login u7 p5 -auth confidential,c1,c2 -bf
        Password:
        ******
        Your authorization is confidential,c1,c2.
        ******
        r 1551 0.760 2.498 47

       *write_tape IO_TEST>OUTPUT reel_id
        1 request signalled, 0 already queued
        r 1553 0.719 12.098 149

  tape: Contents of OUTPUT written on tape reel_id.

TDT-2:  (CONTINUE)
       *new_proc -auth unclassified,c1,c2
        ******
        Your authorization is unclassified,c1,c2.
        ******
        r 1554 0.941 9.554 119

       *write_tape IO_TEST>NO_OUTPUT reel_id
        1 request signalled, 0 already queued
        r 1555 0.258 0.420 23

  tape: No tape is written as a result of this test.
```

```
TDT-3:  (CONTINUE)
       *new_proc -auth confidential,c1
        ******
        Your authorization is confidential,c1.
        ******
        r 1556 0.920 6.806 78

       *write_tape IO_TEST>NO_OUTPUT reel_id
        1 request signalled, 0 already queued
        r 1556 1.817 16.704 123

  tape: No tape is written as a result of this test.

TDT-4:  (CONTINUE)
       *new_proc -auth confidential,c1,c2,c3,c4,c5
        ******
        Your authorization is confidential,c1,c2,c3,c4,c5.
        ******
        r 1557 0.920 6.806 78

       *write_tape IO_TEST>NO_OUTPUT reel_id
        1 request signalled, 2 already queued.
        r 1558 0.218 1.332 33

  tape: No tape is written as a result of this test.

TDT-5:  (CONTINUE)
       *new_proc -auth top_secret,c1,c2
        ******
        Your authorization is top_secret,c1,c2.
        ******
        r 1559 0.921 6.806 78

       *write_tape IO_TEST>NO_OUTPUT reel_id
        1 request signalled, 2 already queued
        r 1560 1.774 16.704 123

  tape: No tape is written as a result of this test.

TDT-6:  (CONTINUE)
       *new_proc -auth confidential,c2,c3
        ******
        Your authorization is confidential,c2,c3.
        ******
        r 1560 0.920 6.806 78
        *write_tape IO_TEST>NO_OUTPUT reel_id
         1 request signalled, 0 already queued.
         r 1561 1.803 17.616 181

   tape: No tape is written as a result of this test.

TDT-7:  (CONTINUE)
        *new_proc -auth confidential,c1,c2
         ******
         Your authorization is confidential,c1,c2.
         ******
         r 1562 0.921 7.482 81

        *change_wdir IO_TEST
         r 1563 0.604 3.234 12

      > *write_tape_test S,1,2>BAD_LEVEL reel_id
         1 request signalled, 2 already queued.
         r 1563 1.817 16.704 153

   tape: Error message indicating inability to access S,1,2>BAD_LEVEL

TDT-8:  (CONTINUE)
      > *write_tape_test C,1,2,3>BAD_CATEGORY reel_id
         1 request signalled, 2 already queued.
         r 1564 1.817 28.020 163

   tape: Error message indicating inability to access C,1,2,3>BAD_CATEGORY

TDT-9:  (CONTINUE)
      > *write_tape_test BAD_ACL reel_id
         1 request signalled 2 already queued.
         r 1564 0.233 2.678 54

   tape: Error message indicating inability to access BAD_ACL

TDT-10: (CONTINUE)
        *read_tape C,1,2>TAPE_INPUT reel_id
         1 request signalled, 0 already queued
         r 1553 0.719 12.098 149

   tape: Tape reel_id read into segment TAPE_INPUT, a segment created
             in directory C,1,2.
```

```
TDT-11: (CONTINUE)
       *new_proc -auth unclassified,c1,c2
        ******
        Your authorization is unclassified,c1,c2.
        ******
        r 1554 0.941 9.554 119

       *read_tape IO_TEST>C,1,2>TAPE_INPUT reel_id
        1 request signalled, 0 already queued
        r 1555 0.258 0.420 23

  tape: No tape is read as a result of this test.

TDT-12: (CONTINUE)
       *new_proc -auth confidential,c1
        ******
        Your authorization is confidential,c1.
        ******
        r 1556 0.920 6.806 78

       *read_tape IO_TEST>C,1,2>TAPE_INPUT reel_id
        1 request signalled, 0 already queued
        r 1556 1.817 16.704 123

  tape: No tape is read as a result of this test.

TDT-13: (CONTINUE)
       *new_proc -auth confidential,c1,c2,c3,c4,c5
        ******
        Your authorization is confidential,c1,c2,c3,c4,c5.
        ******
        r 1557 0.920 6.806 78

       *read_tape IO_TEST>C,1,2>TAPE_INPUT reel_id
        1 request signalled, 2 already queued.
        r 1558 0.218 1.332 33

  tape: No tape is read as a result of this test.

TDT-14: (CONTINUE)
       *new_proc -auth top_secret,c1,c2
        ******
        Your authorization is top_secret,c1,c2.
        ******
        r 1559 0.921 6.806 78
        *read_tape IO_TEST>C,1,2>TAPE_INPUT reel_id
         1 request signalled, 2 already queued
         r 1560 1.774 16.704 123

  tape: No tape is read as a result of this test.

TDT-15: (CONTINUE)
        *new_proc -auth confidential,c2,c3
         ******
         Your authorization is confidential,c2,c3.
         ******
         r 1560 0.920 6.806 78

        read_tape IO_TEST>C,1,2>TAPE_INPUT reel_id
         1 request signalled, 0 already queued.
         r 1561 1.803 17.616 181

  tape: No tape is read as a result of this test.

TDT-16: (CONTINUE)
        *new_proc -auth confidential,c1,c2
         ******
         Your authorization is confidential,c1,c2.
         ******
         r 1562 0.921 7.482 81

        *change_wdir IO_TEST
         r 1562 0.605 2.345 12

      > *read_tape_test S,1,2>BAD_LEVEL reel_id
         1 request signalled, 2 already queued.
         r 1563 1.817 16.704 153

  tape: Error message indicating inability to access S,1,2>BAD_LEVEL

TDT-17: (CONTINUE)
      > *read_tape_test C,1,2,3>BAD_CATEGORY reel_id
         1 request signalled, 2 already queued.
         r 1564 1.817 28.020 163

  tape: Error message indicating inability to access C,1,2,3>BAD_CATEGORY
```

TDT-18: (CONTINUE)

```
    > *read_tape_test C,1,2>BAD_WRITE_ACL reel_id
      1 request signalled 2 already queued.
      r 1564 0.233 2.678 54
```

tape: Error message indicating inability to access BAD_ACL

TDT-19: (CONTINUE)

```
     *print  C,1,2>TAPE_INPUT

            TAPE_INPUT    06/21/75    1565.3  edt  Mon


      This segment contains proper information for output.

      r 1565 0.265 1.222 32

     *delete C,1,2>TAPE_INPUT
      r 1566 0.160 4.686 82
```

OPERATOR: (CLEANUP)

Delete all read_tape and write_tape requests from the tape daemon queues.

SYSTEM SECURITY ADMINISTRATOR (SSA)

The SSA tests as described on page 76 are performed manually by the SSA himself. In the sample script below, it is assumed that the SSA has the user name of SSA and project id of SysAdmin. Reference is made to the directory SSA_TEST, created during initialization.

```
SSA-1: *login SSA -auth confidential
        Password:
        ******
        Your authorization is confidential.
        ******
        r 1200 3.453 12.354 345

       *list_acl >system_library_1>system_privilege_

        re       SSA.SysAdmin.*
        re       *.SysDaemon.*

        r 1201 0.123 1.234 67

        (Note that the ACL may not be exactly like that above, depend-
        ing on who should have access to the system_privilege_ gate.)

SSA-2:  (CONTINUE)
      > *access SSA_TEST
        no privileges
        r 1202 0.123 0.343 23

SSA-3:  (CONTINUE)
       *set_system_priv dir
        r 1202 0.123 0.345 12

      > *access SSA_TEST
        dir
        r 1202 0.012 0.233 45

SSA-4:  (CONTINUE)
       *set_system_priv ^dir seg
        r 1202 0.123 0.345 12

      > *access SSA_TEST
        seg
        r 1202 0.012 0.233 45

SSA-5:  (CONTINUE)
       *set_system_priv ^seg ipc
        r 1202 0.123 0.345 12
        (For this test, the SSA must locate some user logged in at an
        authorization below his own.  Assume the user is u1 on project
        p1.)
       *send_message u1 p1 Hello.
        r 1202 1.089 2.343 45

        (u1.p1 should receive this message, thereby indicating ipc
        privilege is set.)

      > *access SSA_TEST
        no privileges
        r 1202 0.012 0.233 45

SSA-6:  (CONTINUE)
       *set_system_priv ^ipc ring1
        r 1202 0.123 0.345 12

       *send_message u1 p1 Hello
        send_message: Attempt to wakeup a process of a lower authorization.
        r 1202 0.323 5.635 20

        (u1.p1 should not receive this message.)

      > *access SSA_TEST
        ring1
        r 1202 0.012 0.233 45

SSA-7:  (CONTINUE)
       *set_system_priv ^ring1 soos
        r 1202 0.123 0.345 12

      > *access SSA_TEST
        soos
        r 1202 0.012 0.233 45

SSA-8:  (CONTINUE)
       *set_system_priv ^soos
        r 1202 0.123 0.345 12

      > *access SSA_TEST
        no privileges
        r 1202 0.012 0.233 45
       *logout

SSA-9: *login SSA
       *Password:
```

```
       r 1200 1.234 5.678 98

      *create_dir test_dir
       r 1201 2.323 0.564 21

      *mbx_create test_dir>sys_seg
       r 1202 3.424 9.467 23

      *create test_dir>seg
       r 1203 3.424 7.543 90

      *reclassify_sys_seg test_dir>sys_seg confidential
       r 1204 1.234 6.545 78

      *status test_dir>sys_seg -mode

       mode:                null
       ring brackets:       1, 1, 1
       access class:        confidential

       r 1205 1.121 4.345 56

SSA-10: (CONTINUE)
      *reclassify_seg test_dir>seg confidential
       r 1206 1.232 3.432 34

      *status test_dir>seg -mode

       mode:                null
       ring brackets:       4, 4, 4
       access class:        confidential

       r 1207 0.434 1.232 78

SSA-11: (CONTINUE)
      *new_proc -auth confidential
       ******
       Your authorization is confidential
       ******
       r 1208 1.234 56.765 58

      *reclassify_dir test_dir -auth confidential -quota 3
       r 1208 0.232 0.754 45

      *status test_dir -mode
```

```
        mode:                null
        ring brackets:       7, 7
        access class:        confidential

        r 1209 0.121 0.323 33

SSA-12: (CONTINUE)
       *reclassify_dir test_dir confidential -quota 0
        r 1210 1.434 4.323 23

       *list -pn test_dir
        list: There was an attempt to reference a directory which is
             out of service. test_dir
        r 1211 1.212 4.345 34

       *reset_soos test_dir
        reset_soos: The directory is upgraded without terminal quota.
             test_dir
        r 1212 1.232 0.656 45

       *reclassify_dir test_dir confidential -quota 3
        r 1213 1.232 4.234 56

       *reset_soos test_dir
        r 1213 4.323 6.765 45

       *list -pn test_dir

        Segments = 2, records = 0.

        r w     0  seg
                0  sys_seg

        r 1215 3.234 0.233 34

       *logout
```

See Appendix III for a writeup of the access command used to determine which privileges are set.

AUDITING (AUD)

The tests of the audit mechanism described on page 79 are performed by the SSA, or someone who has phcs_ access so that the print_syserr_log command can be used. The test consists of a single call to an exec_com which should invoke each of the audit functions. It is assumed that all of the audit bits are set on for the current user before the user logs in.

```
        *login SSA  -auth S,1,2
         Password:
         ******
         Your authorization is secret, c1, c2
         ******
         r 1209 4.543 12.694 457

      > *exec_com audit DIR SEG

         Segments = 0.

         No mail.

        *Enter name of user logged in below S,1,2: u1

        *Enter his project: p1
         send_message: Attempt to wakeup a process of a lower authorization.
         mail: Mailbox full. >udd>SysAdmin>SSA>SSA.mbx

         SYSERR_LOG FROM 01/01/75  1209.0 edt Mon to 01/01/75 1230.0 edt Mon.

AUD-1:   (not implemented)
AUD-2:   (not implemented)
AUD-3:   (not implemented)
AUD-4:   Incorrect access to [pd]>audit_dir
AUD-5:   Illegal procedure: illegal_procedure by >udd>SysAdmin>SSA>audit
AUD-6:   Access violation: no_write_permission by >udd>SysAdmin>SSA>audit
AUD-7:   Access violation: not_in_read_bracket by >udd>SysAdmin>SSA>audit
AUD-8:   Wakeup denied by SSA.SysAdmin, referencing u1.p1
AUD-9:   System privilege enable: dir
         System privilege disable: dir
AUD-10: Reclassify_dir >udd>SysAdmin>SSA>DIR
AUD-11: (not implemented)
AUD-12: (not implemented)
AUD-13: Message segment overflow referencing >udd>SysAdmin>SSA>SSA.mbx

         r 1239 12.354 4.565 344
```

The arguments DIR and SEG are the names of a directory and segment as described in the writeup of audit.ec in Appendix III.

The exact text of the thirteen audit messages that should appear (labeled AUD-1 to AUD-13) is not shown above because of the numerous variations likely to be encountered in the audit messages, and because the exact nature of the messages has not been completely finalized at the time of this writing. However, the approximate content of each message is indicated. Those messages indicated as not implemented are auditing functions that have not yet been incorporated into Multics. If these functions are incorporated in the future, messages will appear in their place.

The code that invokes each specific auditing function can be found in the listings of the program audit.pl1 and aim_test_exec_coms in Appendix IV.

APPENDIX III

PROGRAM DOCUMENTATION

This appendix contains writeups of all commands, exec_coms and subroutines referenced in the initialization in Appendix I, in the test scripts in Appendix II, or internally by another command or subroutine, that are not part of the standard Multics system as documented in the Multics Programmers' Manual [10]. The writeups labeled "Exec_com" or "Command" are called directly by the user. Those labeled "Active Function" or "Subroutine" are internal interfaces referenced by an exec_com, command, or other subroutine. These writeups appear in alphabetical order. Below is a table showing the correspondence between the writeup in this appendix and the name of the source module containing that program. Appendix IV contains the actual listings of several of these modules explicitly referenced in the scripts or text in Appendix II, or containing explicit code that performs any of the tests. Those modules whose listings are included are indicated by an asterisk in the table.

writeup	source module
access	access.pl1
acl_comparison	acl_comparison.pl1
all	active_functions.pl1
assoc	assoc.pl1
audit	*audit.pl1
audit.ec	*aim_test_exec_coms
authorization_tester	*authorization_tester.pl1
bit_to_integer_	bit_to_integer_.pl1
check_status_	check_status_.pl1
create_test_auth.ec	*aim_test_exec_coms
create_test_dir.ec	*aim_test_exec_coms
create_test_seg.ec	*aim_test_exec_coms
create_test_start_up.ec	*aim_test_exec_coms
diffo_str	diffo_str.pl1
dprint_test, dpunch_test	dprint_test.pl1
encode_authorization	active_functions.pl1
get_callers_ap_	get_callers_ap_.pl1
get_dir_arg_	get_dir_arg_.pl1
goto_seg_	goto_seg_.alm
line_number_inserter	line_number_inserter.pl1
mbx_test.ec	*aim_test_exec_coms
mbx_test_start_up.ec	*aim_test_exec_coms

new_proc_	new_proc_.pl1
new_proc_test	new_proc_test.pl1
number_	number_.pl1
print_acl	print_acl.pl1
process_1_proc	*process_1_proc.pl1
process_2_proc	*process_2_proc.pl1
quota	active_functions.pl1
quota_used	active_functions.pl1
read_tape_test, write_tape_test	read_tape_test.pl1
response_to_start_up	*response_to_start_up.pl1
short_string	active_functions.pl1
terminal_2_proc	*terminal_2_proc.pl1
test_acl_use	test_acl_use.pl1
test_add_list	test_add_list.pl1
test_append_list	test_append_list.pl1
test_delete_list	test_delete_list.pl1
test_dir_auth	*test_dir_auth.pl1
test_ipc	*test_ipc.pl1
test_replace_list	test_replace_list.pl1
test_seg_acl	test_seg_acl.pl1
test_seg_auth	*test_dir_auth.pl1
tipc_set_up	*tipc_set_up.pl1
try_dir_reference_	*try_dir_reference_.pl1
try_reference_	try_reference_.pl1

Name: access

This command determines experimentally which privilege bits are currently set for the process. The following privileges are checked: dir, seg, soos, and ring1. The ipc privilege is not tested, since that can much more easily be checked by using the send_message command. The user need not have access to the system_privilege_ gate to execute this command, but in that case the privileges could probably never have been set.

Usage

```
     access dir
```

1) dir	is the pathname of a special directory that is accessed by this command in order to determine which privileges are set. The contents of this directory is given in Notes below.

Notes

The privilege bits that are set are printed on the terminal. If no privileges are set, the message "no privileges" will appear.

The directory referenced by this command should be an upgraded directory at a higher access class than the current process. It should contain one empty segment named SEG, an empty message segment named MSEG, and an empty subdirectory DIR that is out of service.

Subroutine

Name: acl_comparison

The acl_comparison subroutine, given two segment access control lists, will compare the two ACLs entry by entry to see whether or not they are identical.

Usage

```
   dcl   acl_comparison  entry (1 (*) aligned, 2 char(32), 2 bit(36),
                 2 bit(36), 2 fixed bin(35), 1 (*) aligned, 2
                 char(32), 2 bit(36), 2 bit(36), 2 fixed bin(35),
                 fixed bin(35));

   call   acl_comparison (acl_1, acl_2, code);
```

1) acl_1	is a segment_acl structure as described in the MPM writeup of hcs_$add_acl_entries. (Input)
2) acl_2	is a second segment_acl structure. (Input)
3) code	indicates the results of the comparison. See Notes below. (Output)

Notes

Only the first three components of the ACL structure (group_id, modes and zero_pad) are compared. The status_codes are not compared. The results of the comparison, indicated by the value of code, is as follows:

0	The ACLs are identical.
1	The ACLs have different numbers of entries, and thus cannot be identical.
2	There exists a pair of corresponding entries with different group_id.
3	There exists a pair of corresponding entries with different modes.

4	There exists a pair of corresponding entries with different zero_pad.

Name: all

This active function returns the contents of a segment, with all newlines except the last one changed to blanks.

Usage

```
[all path]
```

1) path   is the pathname of a segment.

Command

Name: assoc

This active function implements an associative memory useful for implementing exec_com variables.

Usage

```
[assoc name]
```

1) name   is a variable which has been set to some value by a prior call to assoc_set.

The returned value is a character string representing the value associated with the supplied name. If the name was not found, a null string is returned.

Name: assoc_set

This entry is used to associate a value with a name of a variable.

Usage

```
assoc_set name1 value1 ... namen valuen
```

1) namei   is a character string of up to 32 characters.

2) valuei is a character string of any length. Blanks contained in the value will be considered part of the value, and will be returned by the assoc active function call. Of course, if there are blanks in the value, the value must be enclosed in quotes. If the valuei is a null string, the associative memory entry for namei will be cleared.

Name: assoc_clear

This entry clears the entire associative memory.

Usage

```
assoc_clear
```

There are no arguments.

Name: assoc_list

This entry lists the associative memory. The value of each entry will be enclosed in quotes (but these quotes are not part of the value) so that the user can determine if there are any leading or trailing blanks as part of the value.

Usage

```
assoc_list
```

There are no arguments.

Name: audit

This command is called by audit.ec to perform several operations that are more easily performed by a program. It should not be called by the user.

Usage

```
     audit option
```

1) option may be one of the following:

no_access	creates a directory in the process directory with null permission and attempts to access it with hcs_$status_.
ipr_fault	attempts to execute a privileged instruction.
acv_mode	creates a segment in the process directory with no write permission and attempts to write into it.
acv_ring	attempts to read the contents of >system_library_1>hcs_.
no_attach	attempts to attach an I/O device.
no_mount	attempts to perform an rcp_ mount.

Notes

The names of the options are the same as the keywords provided to the protection audit commands. Currently the keywords no_mount and no_attach have no effect.

Name: audit.ec

This exec_com tests the protection audit feature of Multics by performing 13 operations, each of which should be audited in the syserr_log. It is assumed that the user executing this exec_com is logged in at some authorization above system_low, and that he has phcs_ access so that he can print the syserr_log. In addition, the user must have access to the system_privilege_ gate.

Usage

```
exec_com audit dir seg
```

1) dir	is the name of some directory with an access class equal to the current authorization. The directory may or may not be empty -- its contents are not affected.
2) seg	is the name of some segment with an access class equal to the current authorization. The segment should contain about 500 characters of ASCII data. The contents of the segment are unaffected.

Notes

Several messages will appear on the terminal when this exec_com is invoked. These messages may be ignored. The user will be asked at one point to enter the name and project of some other user on the system who is logged in below the current authorization. After invoking all the auditing functions, the print_syserr_log command is used to print the syserr_log on the terminal. There should be one message in the syserr_log corresponding to each of the following events, in the following order:

1. Initiate of classified directory dir.
2. Initiate of classified segment seg.
3. Initiate of message segment (user's mailbox).
4. Access denied to a directory "audit_dir" in the process directory.
5. Illegal procedure fault.
6. Access mode violation (no_write_permission) referencing a segment "audit_seg" in the process directory.

7. Ring bracket violation (not_in_read_bracket) referencing >system_library_1>hcs_.
8. Wakeup denied referencing process of user at lower authorization.
9. Enable and disable of directory privilege (2 messages).
10. Reclassification of dir.
11. Attach of I/O device denied.
12. Mount denied.
13. Message segment overflow, referencing user's mailbox.

Command

Name: authorization_tester

This command determines the authorization of the current process (level number and category set) through selected references to classified segments in a special directory. This computed authorization is compared to that supplied by a call to hcs_$get_authorization, and an error message is printed if both are not the same. The special directory is created by the use of the exec_com create_test_auth.ec.

Usage

```
     authorization_tester -dirname- -control_args-
```

1) dirname	is the directory that is assumed to contain the special subtree required by authorization_tester. See Notes below for a description of this subtree. If dirname is not supplied, it will be assumed to be the working directory.

2) control_args may be the following:

-max auth	specifies the maximum authorization to be tested. The default is system_high. The purpose of this argument is to limit the number of directories examined in the special subtree. If the current process authorization is above this specified value, an error message will be printed.

Notes

The directory specified by dirname must contain one subdirectory of each classification level system_low through system_high (no categories), and one subdirectory of each of the categories within the category set of system_high. The classification level of the category directories may be any level below the current process level, but should typically be system_low.

The name of each of these directories is the short name for its authorization, as returned by the subroutine convert_authorization_$to_string_short, followed by the suffix "_auth". Each of the subdirectories also contains a single zero length segment whose name is "seg". This entire subtree can be created by create_test_auth.ec.

The authorization_tester works by trying to initiate segments of successively higher levels, beginning with system_low and ending with the maximum level as specified above. The current level is assumed to be the highest level that was successfully initiated and read. Similarly, the current category set is the logical intersection of all the categories that could be read.

After determining the current level and category set, higher levels and other category sets are read to check that no access is allowed to these. Only read access is actually checked. To check if a "write down" or "write up" is allowed the test_seg_auth command must be used.

Subroutine

Name: bit_to_integer_

This function returns a character string consisting of a series of integers separated by commas that indicate which positions of a bit string have 1´s.

Usage

```
     declare bit_to_integer_ entry (bit(*)) returns (char(*));

     charstring = bit_to_integer_ (bitstring);
```

1) bitstring	is a string of bits. This string may be any length. (Input)
2) charstring	is the string containing integers corresponding to bit positions in bitstring that have 1´s. This string should be long enough to hold the maximum number of integers that are expected. If bitstring is zero in length, or contains no 1´s, the string "none" will be returned.

Example

```
     call ioa_ (bit_to_integer_("1000111"b));
```

will produce the following output:

```
     1,5,6,7
```

Subroutine

Name: check_status_

This subroutine is called by try_dir_reference_ in order to validate the status code returned on each call to an hcs_ entry under test. This subroutine prints an error message if the status code is not as expected.

Entry: check_status_$set

This entry initializes static data consisting mostly of pointers to variables in try_dir_reference_. It is used so that the variables themselves do not have to be passed as arguments on each call to check_status_.

Usage

```
     dcl check_status_$set entry (bit(2), bit(2), ptr, ptr, ptr, ptr,
          label);

     call check_status_$set (mode_tested, mode_expected, code_ptr,
          allowed_code_ptr, not_allowed_code_ptr, reference_ptr,
          return_label);
```

1) mode_tested is either "10"b for modify, or "01"b for status. This argument describes the access mode currently being tested. Its relationship to mode_expected is described in Notes. (Input)

2) mode_expected specifies the access mode that is expected on the directory being tested. The first bit indicates modify, and the second bit indicates status. In addition to either bit, both bits or none (for no access) may be on. Since any one invocation of try_dir_reference_ only checks one directory, this value normally never changes. (Input)

3) code_ptr is a pointer to the status code argument (fixed bin(35)) returned from a previous call to the hcs_ entry being tested. (Input)

4) allowed_code_ptr is a pointer to the status code that will be expected when mode_tested is included in mode_expected. In

other words, if the access mode being tested is allowed, this status code should be returned by the call to the hcs_ entry. For most legitimate calls, there should be no status code, and therefore this argument will point to a zero word. (Input)

5) not_allowed_code_ptr is a pointer to the status code expected when mode_tested is not included in mode_expected. For example, if only "s" permission is expected on a directory under test, and mode_tested is "m", this argument points to the expected code. (Input)

6) reference_ptr is a pointer to a string (char(168)) that describes the item being referenced in the hcs_ call. This string is printed along with any error message. (Input)

7) return_label is a label in try_dir_reference_ to which check_status_ will return in the case of any error. (Input)

Notes

The only function of this entry is to save each of the above pointers and variables for later use by check_status_. Where pointers are specified, only the pointers are saved. The values of the variables pointed to may be freely changed between calls to check_status_. In try_dir_reference_, this entry is called only when mode_tested needs to be respecified, since none of the other items ever change.

Entry: check_status_

This is the entry that is called on each test of an hcs_ entry.

Usage

```
     dcl check_status_ entry options (variable);

     call check_status_ (line_no, entry_no, error_bit, ctl_string,
          ctl_arg1, ..., ctl_argn);
```

1) line_no is the line number (fixed bin) in try_dir_reference_ in which the call to check_status_, or some internal support procedure,

	was made. This line number is printed along with any error message. (Input)
2) entry_no	is an index (fixed bin) into a table of names of hcs_ entry points. The table is shown in <u>Entry Names</u> below. (Input)
3) error_bit	This bit (bit(1)) on indicates that a special message is to be printed even though the proper status code may have been received. If the wrong status code was received, the normal message is printed as described in <u>Notes</u> below. (Input)
4) ctl_string	is the message (char(*)) to be printed when error_bit is on and the proper status code is received. This message is in the form of an ioa_ control string, and may be followed by more arguments for ioa_. (Input)
5) ctl_arg*i*	are possible additional arguments for ioa_. (Input)

<u>Notes</u>

If an error is detected, check_status_ prints a message on user_output and returns to the label specified in check_status_$set. If there is no error, check_status_ just returns from the call.

The error conditions detected are described below. Each condition is described in terms of a logical expression relating the values of several variables. Following the condition, the text of the error message is shown, where the following substitutions should be made for each item enclosed in quotes:

"Status/Modify"	"Status" if mode_tested = "01"b "Modify" if mode_tested = "10"b
"code"	Standard error_table_ message corresponding to the status code pointed to by code_ptr.
"allowed_code"	Message corresponding to the status code pointed to by allowed_code_ptr.

"not_allowed_code"	Message corresponding to the status code pointed to by not_allowed_code_ptr.
"message"	Message composed of the ctl_string and ctl_args passed to check_status_.

1. (mode_expected & mode_tested) = "00"b & (code = 0)

   "Status/Modify" permission was not expected (status code "not_allowed_code" expected), but no status code returned.

2. (mode_expected & mode_tested) = "00"b & (code = not_allowed_code) & error_bit

   "Status/Modify" permission was not expected, and proper status code returned, but "message".

3. (mode_expected & mode_tested) = "00"b & (code ≠ 0) & (code ≠ not_allowed_code)

   "Status/Modify" permission not expected, but status code "code" was returned instead of "not_allowed_code".

4. (mode_expected & mode_tested) ≠ "00"b & (code ≠ allowed_code) & (code = 0)

   "Status/Modify" permission and status code "allowed_code" expected, but none returned.

5. (mode_expected & mode_tested) ≠ "00"b & (code ≠ allowed_code) & (code ≠ 0) & (allowed_code = 0)

   "Status/Modify" permission expected -- status code "code" returned instead.

6. (mode_expected & mode_tested) ≠ "00"b & (code ≠ allowed_code) & (code ≠ 0) & (allowed_code ≠ 0)

   "Status/Modify" permission and status code "allowed_code" expected, but "code" returned instead.

If none of the above conditions occurs, no error message is printed and check_status_ returns. The "no error" condition can be expressed as:

```
     (mode_expected & mode_tested) ≠ "00"b & (code = allowed_code)
  or (mode_expected & mode_tested) = "00"b & (code = not_allowed_code)
        & (error_bit = "0"b)
```

Following any error message, a line of the following form is printed:

```
    Error occurred on a call to "entry_name", referencing "reference"
    (on line "line_no" of try_dir_reference_).
```

where

"entry_name"	is the name of the hcs_ entry corresponding to the entry_no argument.
"reference"	is the reference string passed to check_status_.
"line_no"	is the value of the line_no argument.

Entry: check_status_$return, check_status_$no_return

These two entries turn on or off a flag designating whether check_status_ should return to its caller, or to return_label, in case of an error. The effect of specifying return is to continue testing regardless of errors that are detected. These entries are not called by try_dir_reference_, but from command level by the user. Default is no_return, which says to exit to return_label in case of an error.

Entry: check_status_$debug_on, check_status_$debug_off

These two entries, also called from command level, turn on or off a switch causing check_status_ to print a line of information on every call, instead of only on errors. The line has the following format:

```
    ***("line_no"): "entry_name"  "reference"
```

where "line_no", "entry_name", and "reference" are defined above.

In addition, each time check_status_$set is called, a new page is ejected followed by a line of the following form:

```
*******************"Status/Modify"*****************
```

where "Status/Modify" is as defined above.

Entry Names

The following table lists the number and name of each entry point in hcs_ whose entry number can be passed as the entry_no argument to check_status_.

Name	entry_no
hcs_$add_acl_entries	1
hcs_$add_dir_acl_entries	2
hcs_$add_dir_inacl_entries	3
hcs_$add_inacl_entries	4
hcs_$append_branch	5
hcs_$append_branchx	6
hcs_$append_link	7
hcs_$chname_file	8
hcs_$chname_seg	9
hcs_$del_dir_tree	10
hcs_$delentry_file	11
hcs_$delentry_seg	12
hcs_$delete_acl_entries	13
hcs_$delete_dir_acl_entries	14
hcs_$delete_dir_inacl_entries	15
hcs_$delete_inacl_entries	16
hcs_$fs_get_mode	17
hcs_$fs_get_path_name	18
hcs_$fs_get_ref_name	19
hcs_$fs_get_seg_ptr	20
hcs_$fs_move_file	21
hcs_$fs_move_seg	22
hcs_$fs_search_get_wdir	23
hcs_$fs_search_set_wdir	24
hcs_$get_author	25
hcs_$get_bc_author	26
hcs_$get_dir_ring_brackets	27

hcs_$get_max_length	28
hcs_$get_process_usage	29
hcs_$get_ring_brackets	30
hcs_$get_safety_sw	31
hcs_$get_search_rules	32
hcs_$initiate	33
hcs_$initiate_count	34
hcs_$initiate_search_rules	35
hcs_$list_acl	36
hcs_$list_dir_acl	37
hcs_$list_dir_inacl	38
hcs_$list_inacl	39
hcs_$make_ptr	40
hcs_$make_seg	41
hcs_$quota_get	42
hcs_$quota_move	43
hcs_$replace_acl	44
hcs_$replace_dir_acl	45
hcs_$replace_dir_inacl	46
hcs_$replace_inacl	47
hcs_$reset_working_set	48
hcs_$set_bc	49
hcs_$set_bc_seg	50
hcs_$set_dir_ring_brackets	51
hcs_$set_max_length	52
hcs_$set_max_length_seg	53
hcs_$set_ring_brackets	54
hcs_$set_safety_sw_seg	55
hcs_$set_saftey_sw	56
hcs_$star_	57
hcs_$star_list_	58
hcs_$status_	59
hcs_$status_long	60
hcs_$status_minf	61
hcs_$status_mins	62
hcs_$terminate_file	63
hcs_$terminate_name	64
hcs_$terminate_noname	65
hcs_$terminate_seg	66
hcs_$truncate_file	67
hcs_$truncate_seg	68

hcs_$wakeup	69
hcs_$create_branch_	70
hcs_$get_access_class	71
hcs_$get_access_class_seg	72

Subroutine

Name: conv_

This procedure returns a character string representation of certain PL/I data types.

Entry: conv_$fb

This entry returns the character representation of a fixed bin(35) number.

Usage

```
dcl conv_$fb entry (fixed bin(35)) returns (char(20));

string = conv_$fb (n);
```

1) n	is the number whose value is to be converted. (Input)
2) string	is the number represented as a character string, left justified. If the value of n is -1, this string will have the value "not returned".

Entry: conv_$ptr

This entry returns the value of a pointer.

Usage

```
dcl conv_$ptr entry (ptr) returns (char(20));
string = conv_$ptr (ptr);
```

1) ptr	is the pointer to be converted. (Input)
2) string	is the value of the pointer, in the format as produced by the "^p" specification of ioa_. If the pointer is null, the string "not returned" is returned.

Name: create_test_auth.ec

This exec_com creates the subtree required for the authorization_tester command.

Usage

```
exec_com create_test_auth path "(class1 ... classn)"
```

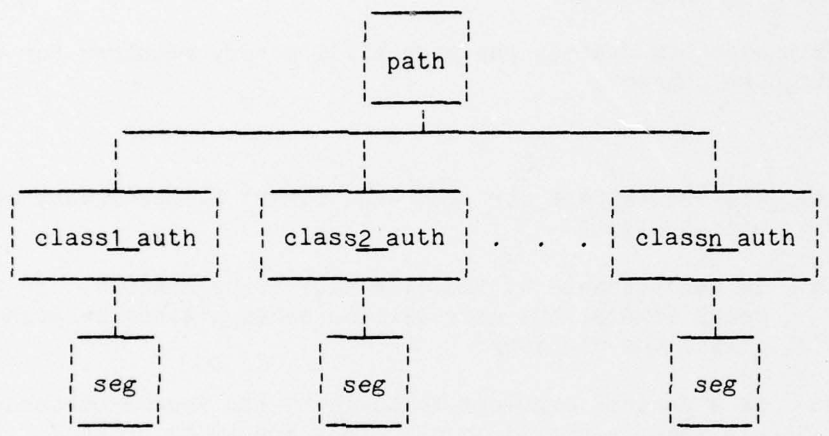
1) path — is the pathname of the directory to be created, in which directories of various access classes will appear.

2) classi — are the names of each of the levels and each of the categories within system_high (except system_low), separated by spaces, and enclosed in parentheses and quotes as shown. The order is unimportant. The user must be able to new_proc to each of these authorizations.

Notes

The operation of this exec_com is very similar to the operation of create_test_dir.ec and create_test_seg.ec. See the writeup of create_test_dir.ec for a description.

The subtree created is illustrated on the next page.


path
class1_auth
class2_auth
. . .
classn_auth
seg
seg
seg

Name: create_test_dir.ec

This exec_com creates the special directory required for the test_dir_auth command.

Usage

```
     exec_com create_test_dir path -acc class1 class2 class3 class4
          class5 class6
```

1) path　is the pathname of the directory to be created. If it already exists, the user will be asked whether he wishes to delete the old copy.

2) -acc　is a control argument followed by the access classes of the six subdirectories in path that are to be created. These access classes may be any values that have a specific relationship to the authorization at which test_dir_auth is to be run. See Notes below.

Notes

If the home directory contains a segment named "create_test_acl", that segment is assumed to contain a list of access identifiers, one per line, that are to be placed on the ACLs of the segments and directories created by this exec_com. If that segment does not exist, only the current user will be given access. It is important to realize that the test_dir_auth command will only operate properly if the user is on the ACL of all the directories and segments created by this exec_com. The actual access modes should not be specified in create_test_acl.

This exec_com creates upgraded directories at the six access classes by performing new_procs to each of the authorizations represented by the classi arguments. In order for this exec_com to work properly, the exec_com create_test_start_up.ec should be called at each of these new_procs. Such a call can be safely placed in the user's start_up.ec (to be executed at new_proc time) because it will have no effect unless create_test_dir.ec was called last in the previous process, or if the process is at system_low. The user may quit any time during the operation of these exec_coms. If the operation is to be aborted, a manual new_proc to system_low will return the user to his original working directory.

The call to create_test_dir.ec must be made from a process currently at system_low. Several temporary segments are created in the user's home directory that are used to drive create_test_start_up.ec for the subsequent processes. One of these segments, called "who", contains the name of the original exec_com that was called (in this case "create_test_dir") and is read by create_test_start_up.ec to determine what operations to perform. If this segment "who" is not found, no action will be performed. After all the directories have been created, the temporary segments will be deleted and the process will be restored to system_low in the original working directory from which create_test_dir.ec was called.

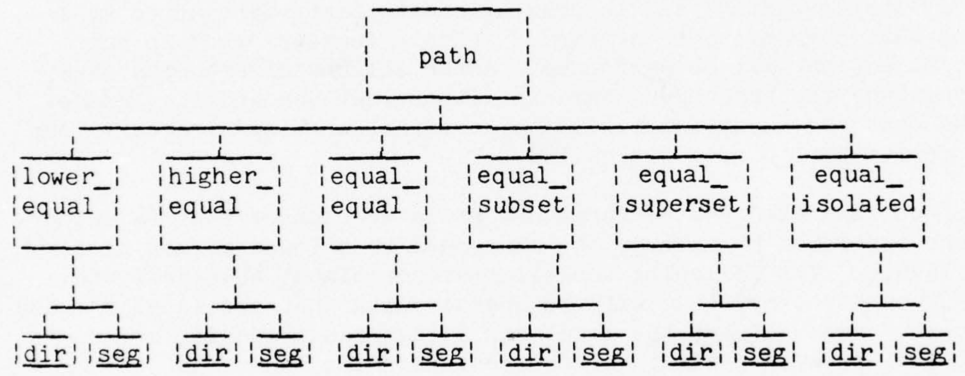
Since this exec_com performs new_procs to each of the six authorizations specified by class*i*, the user must be allowed to new_proc to these levels. The following table lists each class*i* argument, the name of the directory that will be created with that access class, and the relationship between the level and category set specified for class*i* and the authorization of the process which is to run test_dir_auth.

	level	category set	directory name
class1	lower	equal	lower_equal
class2	higher	equal	higher_equal
class3	equal	equal	equal_equal
class4	equal	subset	equal_subset
class5	equal	superset	equal_superset
class6	equal	isolated	equal_isolated

The table above indicates which access class is to be specified in the command line for each class*i* argument. For example, the value of class4 should be an access class that has an equal level and whose category set is a subset of the authorization of the process that will be running test_dir_auth.

If, when create_test_dir.ec is first invoked, the directory specified by path already exists, the user will be asked whether he wishes to delete it. This deletion may fail if the directory contains non-empty upgraded directories. Such directories must be deleted manually by a process of the proper authorization. Alternatively, system privileges "dir" and "seg" can be set (if the user has access to the system_privilege gate), and this exec_com can be called to delete the previously existing directory.

The structure of the subtree created by this exec_com is illustrated below:



Name: create_test_seg.ec

This exec_com creates a special directory that is required for the test_seg_auth command.

Usage

```
exec_com create_test_seg path -acc class1 class2 class3 class4
     class5 class6
```

1) path is the path name of the directory to be created. If it already exists, the user will be asked whether he wishes to delete the old copy.

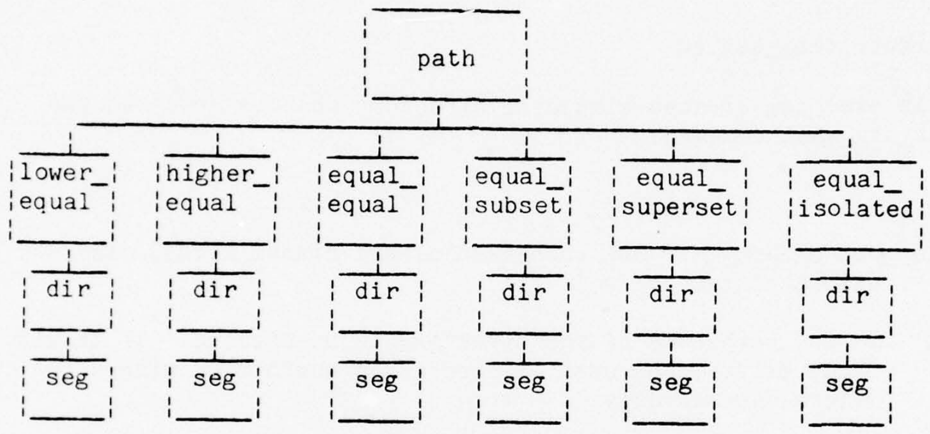
2) -acc is a control argument which must appear, followed by six access class arguments. The six access classes are the access classes to be assigned to the six subdirectories within path, and have a specific relationship to the authorization at which the test_seg_auth command is to be run. See Notes below.

Notes

The operation of this exec_com is very similar to the exec_com create_test_dir.ec. The exec_com create_test_start_up.ec should be called after each new_proc performed by this exec_com. In addition, the segment "create_test_acl" in the home directory is accessed to obtain the names for the ACL of the directories and segments to be created. For a description of the operation, and the six class*i* arguments, see the writeup of create_test_dir.ec.

In addition to the requirements specified for create_test_dir.ec, this exec_com expects a segment named test_seg_auth_ to exist in the working directory. The contents of this segment is required to fill in the first few words of the segments created by create_test_start_up.ec on behalf of this exec_com and required by test_seg_auth.

The structure of the subtree created by this exec_com is illustrated below:



The contents of each segment named "seg" will be the contents of the segment named "test_seg_auth_" obtained from the original working directory.

Name: create_test_start_up.ec

This exec_com is called by the user immediately after each new_proc forced by the use of create_test_dir.ec, create_test_seg.ec or create_test_auth.ec. Calling it separately will usually have no effect.

Usage

```
exec_com create_test_start_up
```

Notes

When called, this exec_com decides what to do based on information contained in various segments in the home directory and on the current authorization. A description of each of these segments follows:

1) who contains the name of the original exec_com that was called, either "create_test_dir, "create_test_seg", or "create_test_auth".

2) original_wdir is the pathname of the working directory from which the original exec_com was called.

3) pathname is the pathname argument to the original exec_com.

4) ac_names is a string of the form:

$class_1$dirname_1$...$class_n$dirname_n$

where each pair $class_i$dirname_i$ is composed of the class_i argument supplied to the original exec_com, transformed into a short string, and the name of the directory that was created with that access class, as defined by the particular exec_com used.

After each new_proc, the process authorization is examined. If it is system_low, the temporary segments above (if they exist) are deleted and the original working directory (if specified) will be restored. If not system_low, the segment "who" is examined. If it does not exist, no operation is performed. If it exists, the process authorization should equal one of the class*i*´s in the "ac_names" segment. The corresponding directory will then be filled in as required by the exec_com specified in "who", and another new_proc to the next class*i* will be performed. The last new_proc will be to system_low, which will cause the temporary segments to be deleted.

Subroutine

Name: diffo_str

The diffo_str subroutine, given five character strings, will select the first of the last three that is different from both the first two. In other words, if you have two character strings and want a string that is different from them both, then you must supply three candidate strings. This subroutine will then pick the first one of the candidates that is different from both of your two original strings.

Usage

```
    dcl diffo_str  entry (char(*), char(*), char(*), char(*), char(*),
                   char(*), fixed bin(35));

    call diffo_str (str_1, str_2, str_3, candidate_1, candidate_2,
                   candidate_3, code);
```

1) str_1	is a character string. (Input)
2) str_1	is a character string. (Input)
3) str_3	If code is 0, then this is the first of the strings candidate_1, candidate_2, candidate_3 that is different both from str_1 and str_2 above. If code is not 0, then this value is undefined. See Status Code Values below. (Output)
4) candidate_1	is a character string of length less than or equal to the length of str_3 above. (Input)
5) candidate_2	See candidate_1 above. (Input)
6) candidate_3	See candidate_1 above. (Input)
7) code	is a standard status code. See Status Code Values below. (Output)

<u>Status Code Values</u>

The values mean the following:

0	The string that is different both from str_1 and str_2 is contained in str_3.
1	The length of either candidate_1, candidate_2, or candidate_3 is greater than the length of str_3.
2	Unsuccessful, probably the three candidate strings were not three different strings.

Command

Name: dprint_test, dpunch_test

These commands operate exactly like dprint and dpunch, except that no check is made if the user or SysDaemons have no access to the segment or containing directory, or if the segment is not found. The dprint or dpunch request is always queued.

Active Function

Name: encode_authorization

This active function returns the encoded form of an authorization string, as provided by convert_authorization_$encode.

Usage

```
[encode_authorization auth_string]
```

1) auth_string is an authorization string. It must be enclosed in quotes if it contains any blanks.

Notes

Note that this active function may return a null string for "system_low", as returned by convert_authorization_$encode.

If the auth_string is invalid, the string "**" is returned.

Subroutine

Name: get_callers_ap_

This subroutine returns a pointer to the argument list of the caller's caller, i.e., if A calls B and B calls C, then C can get a pointer to the argument list that A passed to B by calling get_callers_ap_. If C wants a pointer to its own argument list (the one passed to it by B), it can call cu_$arg_list_ptr.

Usage

```
declare get_callers_ap_ entry returns (pointer);

ap = get_callers_ap_ ();
```

1) ap   is a pointer to the argument list of the caller's caller.

Subroutine

Name: get_dir_arg_

This subroutine gets an argument from the caller's argument list and, assuming the argument is the pathname of a directory, returns the full pathname of the directory. If no argument is found, the pathname of the working directory is returned.

Usage

```
declare get_dir_arg_ entry (fixed bin, char(*), fixed bin(35));

call get_dir_arg_ (argno, dirpath, code);
```

1) argno	is the number of the argument expected to be a directory name. (Input)
2) dirpath	is the full absolute pathname of the directory. If the argument selected by argno does not exist, the pathname of the working directory will be returned. If the argument was bad (in case of a nonzero status code below), the argument itself, or the resulting pathname, is returned. (Output)
3) code	is a standard status code. It will be zero if there was no argument or if the argument was the name of a valid directory. If nonzero, the argument did not point to a directory that exists. (Output)

Subroutine

Name: goto_seg_

This subroutine merely transfers to a location, given a pointer. It can be used to transfer control to another subroutine by using a transfer instruction instead of a call (callsp or call6) instruction.

Usage

```
     declare goto_seg_ entry (bit(36) aligned, ptr);

     call goto_seg_ (word, entryptr);
```

1) word — is a word of data to be passed as an argument to the procedure being called. (Input/Output)

2) entryptr — is a pointer to the entry point to be called. When the subroutine invoked by the transfer to entryptr returns, return will be directly to the statement after this call to goto_seg_ (i.e., the effect will be the same as if the entryptr had been called directly.) The goto_seg_ subroutine has no stack frame of its own, so calls to it will be transparent. (Input)

line_number_inserter   line_number_inserter

Subroutine

Name: line_number_inserter

This command is used to "patch" certain lines in the source of the programs try_dir_reference_.pl1 and test_dir_auth.pl1, so that error conditions can be properly reported by these procedures when they are executed. Within these programs, there are calls to various error handling subroutines, and the first argument to these subroutines is the source line number from which the call was made. The error handling routines can then report the line number to the user as an aid in locating the cause of the error in the source program. Since PL/I provides no facility for passing the line number as an argument automatically, these line numbers must be passed as constants. The line_number_inserter is run on try_dir_reference_.pl1 and test_dir_auth.pl1 to insert the proper line numbers each time these programs are changed.

Usage

```
line_number_inserter path
```

1) path	is the pathname of the source program to be patched. This should be try_dir_reference_.pl1 or test_dir_auth.pl1, or may be any other PL/I source programs to be patched in the manner described in Notes below. The updated source replaces the original.

Notes

This command searches through the segment specified for an exact match with any of the following strings:

```
"call check_status_ ("
"call set_acl_test ("
"call set_saved_loc ("
"call list_acl_test ("
```

Only strings exactly as above will be considered a match -- more or fewer spaces between the words will cause the match to fail.

The four characters following each matching string in the original source segment must form an integer constant (leading blanks permitted within the four character field) and the character after the integer must be a comma. If this is the case, the existing integer constant is replaced by another constant equal to the line number of that statement within the segment.

An error will be detected if a match is found but the integer and comma do not follow as required. If an error occurs, the source will have been modified up to that line.

Example

Assume line 245 in the source program appeared as follows:

```
/* Example */  call set_saved_loc (    1, "DSC-13");
```

After running line_number_inserter, the line will be changed to the following:

```
/* Example */  call set_saved_loc ( 245, "DSC-13");
```

Note that the field width allows up to 9999 lines in the source segment.

Name: mbx_test.ec

This exec_com performs the message segment tests of the access isolation mechanism utilizing the mail command. It new_procs itself to various authorizations while sending messages to the user's mailbox. Then, at a fixed authorization, it attempts to read the messages. The user then can determine whether the expected messages appear. In order to use this exec_com, the user must place a call to mbx_test_start_up.ec in his start_up.ec to be executed at new_proc time.

Usage

```
exec_com mbx_test class1 class2 class3 class4 class5 class6
```

1) class*i* are six authorizations that the user is allowed to new_proc to. These six authorizations must have a specific relationship to each other which is the same as those for create_test_dir.ec. The authorization class3 may be any authorization above system_low that contains at least two categories.

Notes

After validating the arguments, the user will be asked whether he wishes to delete his old mailbox. The mailbox will not be deleted if it contains mail. Then, several temporary driving segments are created in the home directory in a manner similar to create_test_dir.ec, and the new_proc to the six authorizations begins. From that point on, the operation of the command is self explanatory.

Name: mbx_test_start_up.ec

This exec_com is called at each new_proc performed by mbx_test.ec. Its operation is very similar to the exec_com create_test_start_up.ec. The difference is that, instead of creating directories or segments at each new_proc, this exec_com sends a message to the user's mailbox.

Usage

```
exec_com mbx_test_start_up
```

Notes

See the writeup of create_test_start_up.ec.

Subroutine

Name: new_proc_

The new_proc_ subroutine creates a new process with possibly a new access authorization.

Usage

```
   declare new_proc_  entry (bit(72) aligned, fixed bin(35));

   call new_proc_ (new_auth, code);
```

1) new_auth	is the internal form of a standard Multics access authorization. (Input)
2) code	is a standard system status code. (Output)

Notes

The only way that one will return from this subroutine is when an error has occurred in its execution. That error will be be returned in "code".

<u>Name</u>: new_proc_test

This command operates just like the Multics new_proc command, except that no checks are made for a valid -authorization argument. Thus, a new_proc will always take place, and any errors in the authorization argument should be detected by the answering service. See the MPM writeup of new_proc for description of the arguments.

Subroutine

Name: number_

This function returns the character string representation of the decimal value of binary integer. It is exactly like the char builtin function of PL/I, except that the string returned has leading blanks stripped.

Usage

```
declare number_ entry (fixed bin) returns (char(*));

charstring = number_ (n);
```

1) n	is the number to be represented as a character string. (Input)
2) charstring	is the character string representation of the number in decimal. The length of this string will be the minimum number of characters necessary to represent the number, i.e., there will be no blanks in this string.

Subroutine

Name: print_acl

The print_acl subroutine, given a segment Access Control List (ACL), will print it out on "user_output". There will be normal ACL ordering and everything will appear left-justified as illustrated by the following example:

```
rew       Jones.Sys.*
e         Smith.*.*
rw        *.SysDaemon.*
null      *.Beta.*
```

Usage

```
   dcl   print_acl  entry (1 (*) aligned, 2 char(32), 2 bit(36), 2
                   bit(36), 2 fixed bin(35), fixed bin(35));

   call   print_acl (acl, code);
```

1) acl	is a segment_acl structure. See Notes below. (Input)
2) code	is a standard status code. See Notes below. (Output)

Notes

The following structure is used:

```
     dcl 1 segment_acl (*) aligned,
             2 group_id char(32),
             2 modes bit(36),
             2 zero_pad bit(36),
             2 status_code fixed bin(35);
```

1) group_id	is the group identifier (in the form person.project.tag) which identifies the processes to which this acl entry applies.
2) modes	contains the access modes for this group identifier. The first three bits correspond to the access modes read, execute, write. The remaining bits must be zero.

3) zero_pad	must contain zero. (This field is used for extended access)
4) status_code	is a standard status code for only this entry.

Certain errors are looked for in the input segment ACL. If found, a nonzero code will be returned and the ACL will not print out on "user_output". The two errors that are looked for and returned in code when detected are the errors "empty_acl", and "bad_acl_mode". No attempt is made to detect the possible ACL error "bad_name", and thus bad group_ids will print out when input to print_acl.

Subroutine

Name: process_1_proc

The process_1_proc subroutine is called by the test_seg_acl command when invoked from the first terminal. (See the writeup of the test_seg_acl command.) It does the following:

1) Creates the temporary directory test_seg_acl_workspace_dir in the process directory of u1.

2) Creates the temporary segment test_seg_acl_mailbox in the directory >udd>p1>u1.

3) Places IPC information in the segment test_seg_acl_mailbox.

4) Directs u1 to go to a terminal t2 and issue the second form of the test_seg_acl command.

5) Waits for the second form of the test_seg_acl command to fill the segment test_seg_acl_mailbox with IPC information.

6) Activates five modules, which test the basic access control mechanism.

7) Upon completion, cleans up temporary work space and causes the termination of the second form of the test_seg_acl command.

Usage

```
   declare process_1_proc  entry (label);

   call process_1_proc (abandon_test_seg_acl);
```

1) abandon_test_seg_acl	is the label constant affixed to the statement terminating the test_seg_acl command. (Input)

Subroutine

Name: process_2_proc

The process_2_proc subroutine is called by the test_seg_acl command when invoked from the second terminal. (See the writeup of the test_seg_acl command.) It does the following:

1) Fills the segment >udd>p1>u1>test_seg_acl_mailbox with IPC information.

2) Waits for instructions from process P1 to make various access attempts on the segment try_me.

3) Communicates the results of the above access attempts back to P1.

4) Upon termination, cleans up any temporary work area.

Usage

```
declare process_2_proc  entry (label, char(*), char(*));

call process_2_proc (abandon_test_seg_acl, u1, p1);
```

1) abandon_test_seg_acl	is the label constant affixed to the statement terminating the test_seg_acl command. (Input)
2) u1	is the Multics user name. See writeup of the test_seg_acl command. (Input)
3) p1	is the Multics project id. (Input)

Active function

Name: quota

This active function returns the quota of a directory.

Usage

[quota path]

1) path   is the pathname of a directory.

<u>Name</u>: quota_used

This active function returns the quota used of a directory.

<u>Usage</u>

```
[quota_used path]
```

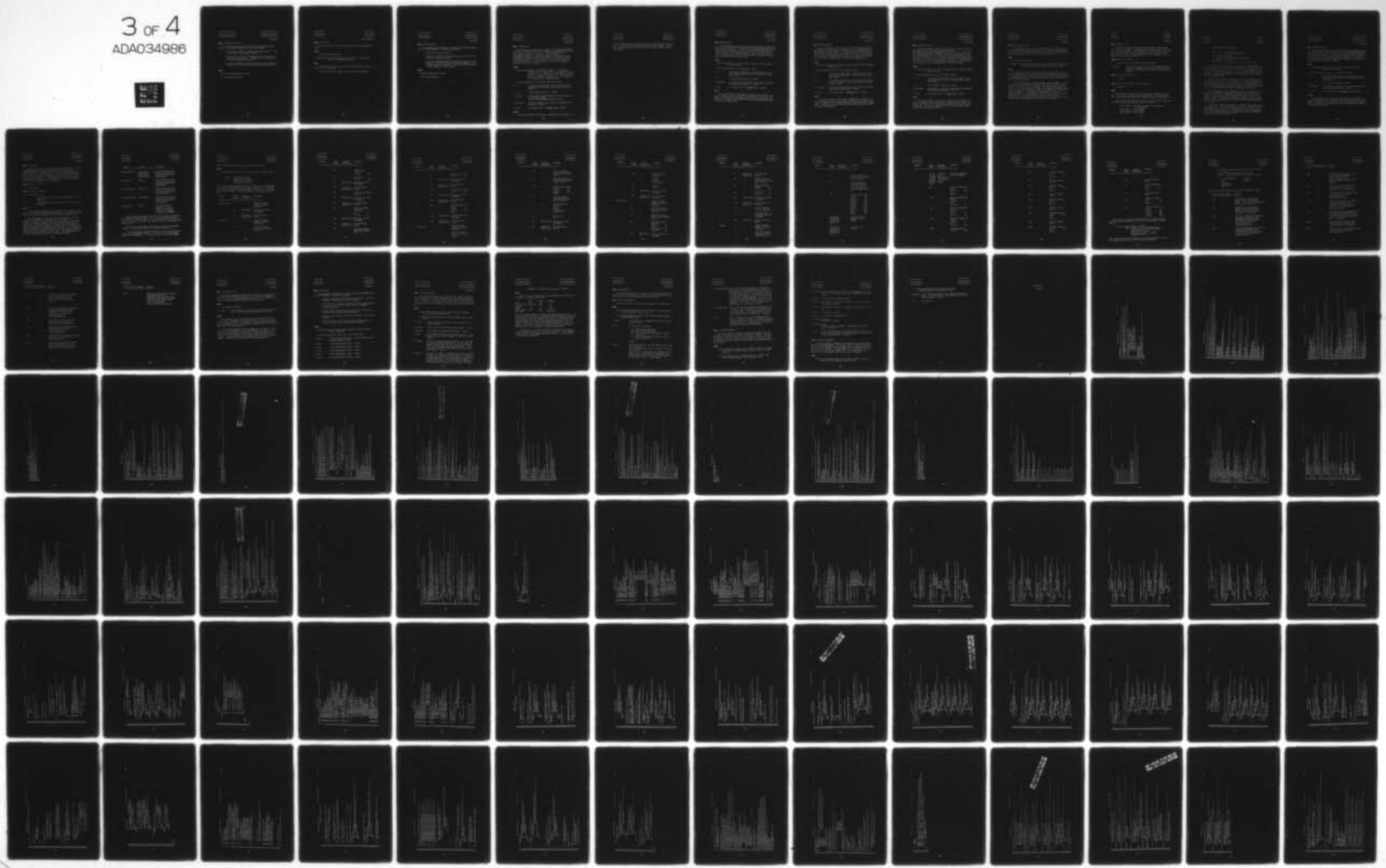
1) path   is the pathname of a directory.

Names: read_tape_test, write_tape_test

These commands are identical to the system read_tape and write_tape commands, except that no check is made to see if the user has the proper access to the segment specified, or if the segment is not found. The request is always queued.

AD-A034 986 MITRE CORP BEDFORD MASS F/G 15/3
TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS,(U)
DEC 76 M GASSER, S R AMES, L J CHMURA F19628-76-C-0001
UNCLASSIFIED MTR-3005 ESD-TR-76-164 NL

3 OF 4
ADA034986



Subroutine

Name: response_to_start_up

The response_to_start_up subroutine is called by the test_ipc command from the first terminal. It does the following:

1) Initiates the temporary segment "multi_process_info". (See the writeup of the test_ipc command.)

2) Determines the correct IPC message to send to the process using the second terminal. Determines the authorization of the "next" process to new_proc to.

3) Sends the message to the process using the second terminal. New_procs to a process with correct "next" authorization.

Usage

```
declare response_to_start_up entry;

call response_to_start_up;
```

Active Function

Name: short_string

This active function returns the short form of an authorization string.

Usage

```
[short_string auth_string]
```

1) auth_string is an authorization string. It must be enclosed in quotes if it contains any blanks.

Notes

Note that a null string may be returned for unnamed authorizations, such as system_low.

If the auth_string is invalid, the string "**" is returned.

Name: terminal_2_proc

The terminal_2_proc subroutine is called by the test_ipc command from the second terminal. It does the following:

1) Creates an event wait channel.

2) Outputs on "user_output" its process_id and the id of the event wait channel it created.

3) Looks for messages sent from the processes using the first terminal. (See the writeup of the test_ipc command.) Any messages that are received but should not be are noted with an error message on "user_output".

Usage

```
declare terminal_2_proc entry;

call terminal_2_proc;
```

Name: test_acl_use

The test_acl_use subroutine is called by the process_1_proc subroutine to test that the ACL of the segment try_me restricts correctly the access of process P2 to try_me. (See the writeup of the test_seg_acl command.) It uses the hcs_$append_branch, hcs_$add_acl_entries, hcs_$delete_acl_entries, and hcs_$replace_acl subroutines in making a series of alterations on the ACL of try_me. After each alteration, it instructs P2 on t2 to access try_me. It verifies that the P2 access to try_me was restricted in accordance with the current ACL of try_me.

Usage

```
   declare test_acl_use  entry (char(*), char(*), 1, 2 bit(36)
                 aligned, 2 char (168) aligned, 2, 3 fixed bin(71), 3
                 fixed bin(71), 3 bit(36), 2, 3 fixed bin(71), 3 fixed
                 bin(71), 3 bit(36), 2 char(32), 2, 3 fixed bin(35), 3
                 char(32), 3 bit(36), 3 bit(36), ptr, 1, 2 fixed bin,
                 2 (1) fixed bin(71), fixed bin(35));

   call test_acl_use (u1, p1, mailbox, wait_list, code);
```

1) u1	is the Multics user name. See the writeup of the test_seg_acl command and process_1_proc subroutine. (Input)
2) p1	is the Multics project id. (Input)
3) mailbox	is the content of the IPC mailbox which resides in the temporary segment >udd>p1>u1>test_seg_acl_mailbox. (Input)
4) wait_list	is the one element list of event wait channels for process P1. (Input)
5) code	is a status code. See Notes below. (Output)

Notes

The value returned in code above is either zero or non-zero. If zero is returned, then no errors were encountered in the test that the ACL of a segment restricts access correctly to that segment. If non-zero, then some sort of error occurred and was noted on the stream "user_output".

Subroutine

Name: test_add_list

The test_add_list subroutine is called by the process_1_proc subroutine to test the mutual consistency of the hcs_$add_acl_entries and hcs_$list_acl subroutines. It calls the hcs_$add_acl_entries subroutine a series of times, each time attempting to add certain ACL entries to the ACL of the segment add_list. After each add attempt, it calls the hcs_$list_acl subroutine and compares the listed ACL with the ACL expected after the add attempt.

Usage

```
    declare test_add_list  entry (char(*), char(*), char(*) aligned,
                  fixed bin(35));

    call test_add_list (u1, p1, path_name, code);
```

1) u1	is the Multics user name. See the writeup of the test_seg_acl command and the process_1_proc subroutine. (Input)
2) p1	is the Multics project id. (Input)
3) path_name	is the Multics path name for the temporary directory test_seg_acl_workspace_dir. (Input)
4) code	is a status code. See Notes below. (Output)

Notes

The value returned in code above is either zero or nonzero. If zero is returned, then no errors were encountered in the test of the mutual consistency of the hcs_$add_acl_entries and hcs_$list_acl subroutines. If nonzero, then some sort of error occurred and was noted on the stream "user_output".

Subroutine

Name: test_append_list

The test_append_list subroutine is called by the process_1_proc subroutine to test the mutual consistency of the hcs_$append_branch and hcs_$list_acl subroutines. It creates the segments append_list_1, append_list_2, append_list_3, and append_list_4 in the temporary directory test_seg_acl_workspace_dir. After the creation of each segment, it calls the hcs_$list_acl subroutine and compares the listed ACL with the ACL expected after the call to hcs_$append_branch.

Usage

```
declare test_append_list  entry (char(*), char(*), char(*) aligned,
                 fixed bin(35));

call test_append_list (u1, p1, path_name, code);
```

1) u1	is the Multics user name. See the writeups of the test_seg_acl command and the process_1_proc subroutine. (Input)
2) p1	is the Multics project id. Again, see the writeups of the test_seg_acl command and process_1_proc subroutine. (Input)
3) path_name	is the Multics path name for the temporary directory test_seg_acl_workspace_dir. (Input)
4) code	is a status code. See Notes below. (Output)

Notes

The value returned in code above is either zero or nonzero. If zero is returned, then no errors were encountered in the test of the mutual consistency of the hcs_$append_branch and hcs_$list_acl subroutines. If nonzero, then some sort of error occurred and was noted on the stream "user_output".

Subroutine

Name: test_delete_list

The test_delete_list subroutine is called by the process_1_proc subroutine to test the mutual consistency of the hcs_$delete_acl_entries and hcs_$list_acl subroutines. It first uses the hcs_$add_acl_entries subroutine to construct a sizeable ACL for the segment delete_list. It then calls the hcs_$delete_acl_entries subroutine a series of times, each time attempting from delete certain ACL entries from the ACL of the segment delete_list. After each delete attempt, it calls the hcs_$list_acl subroutine and compares the listed ACL with the ACL expected after the deletion attempt.

Usage

```
   declare test_delete_list  entry (char(*), char(*), char(*) aligned,
                  fixed bin(35));

   call test_delete_list (u1, p1, path_name, code);
```

1) u1	is the Multics user name. See the writeup of the test_seg_acl command and the process_1_proc subroutine. (Input)
2) p1	is the Multics project id. (Input)
3) path_name	is the Multics path name for the temporary directory test_seg_acl_workspace_dir. (Input)
4) code	is a status code. See Notes below. (Output)

Notes

The value returned in code above is either zero or nonzero. If zero is returned, then no errors were encountered in the test of the mutual consistency of the hcs_$delete_acl_entries and hcs_$list_acl subroutines. If nonzero, then some sort of error occurred and was noted on the stream "user_output".

Command

Name: test_dir_auth, tda

This command utilizes a special test directory to check that the access isolation controls work properly with respect to directories. In order to use this command properly, the special test directory must first be created using the exec_com "create_test_dir.ec".

Usage

```
    test_dir_auth -dirname-
```

1) dirname	is the pathname of the special test directory described below. If missing, the working directory is used.

Notes

The authorization of the process calling this command must be a certain level and category set combination as specified in the writeup to create_test_dir.ec. The user does not need system privilege to run this command -- in fact, he probably shouldn't have it if the controls are to be tested properly.

If the test succeeds, no error messages will be printed. If the test fails, a message will be printed indicating the reason for the failure (bad status code or condition), the expected status code or condition, and the directory or segment that was being referenced when the error occurred. The access class of the segment can be determined from the segment's or directory's pathname (see the writeup of create_test_dir.ec). A comprehensive discussion of most of the error messages that can be produced may be found in the writeup of check_status_. There are, however, additional errors that may turn up that will produce messages not discussed in that writeup.

Command

Name: test_ipc, tipc

The test_ipc command tests that the interprocess communication (IPC) facility of Multics is working correctly with respect to the access isolation mechanism. The command must be issued twice, once from each of two terminals, and then repeatedly at each new_proc on the first terminal. The different invocations of the command are distinguished by the number of arguments.

Usage (from first terminal)

```
test_ipc  auth1 auth2 auth3 auth4 auth5 auth6
```

1) auth*i*	are the names of six Multics access authorizations as specified in Notes below. The user must be able to new_proc to a process having any one of these authorizations.

Usage (from second terminal)

```
test_ipc
```

Usage (from first terminal, at each new_proc)

```
test_ipc  -go
```

Notes

The command is called at the first terminal to begin the test of IPC. The user must be logged in at "system_low". For later reference, the terminal from which this command is issued is known as "t1".

The six auth*i* arguments are six Multics authorizations whose level numbers and category sets are related as follows:

```
level (auth3) = any level not equal to "system_low"
                or "system_high"
level (auth1) < level (auth3)
level (auth2) = level (auth3)
level (auth4) = level (auth3)
level (auth5) > level (auth3)
level (auth6) = level (auth3)

cat (auth3) = any category set having
              at least two components
cat (auth1) = cat (auth3)
cat (auth2) = a proper subset of cat (auth3)
cat (auth3) = a proper subset of cat (auth4)
cat (auth5) = cat (auth3)
cat (auth6) = a category set isolated from cat (auth3)
```

Upon receiving instructions to do so, the user must then call test_ipc from a second terminal without arguments. This second terminal is known as "t2". The user must login at this terminal at an authorization equal to auth3 above. It is not important whether he uses the same or different name and project as used on the first terminal (although a system parameter may be set that does not allow multiple logins by the same user).

After the user calls the command from t2, test_ipc performs several new_procs at t1. After each new_proc, the user must continue the operation of the command by calling test_ipc with the argument -go. It is recommended that the user's start_up.ec be modified for the test to call this command automatically at new_proc. The call should have no effect unless test_ipc was called in the previous process.

For each new_proc on t1, a message is sent to the process at t2 using the IPC facility. Beginning at system_low, the first new_proc creates a process with authorization equal to auth1. The second new_proc creates a process with authorization equal to auth2. This continues until a last new_proc destroys a process with authorization equal to auth6, and creates a final process with authorization equal to "system_low".

The test_ipc command called from t2 looks for the message sent by the process at t1. If it receives one sent from a t1 process having authorization auth4, auth5, or auth6, its prints an error on t2. If no violations are detected, ready messages will print out on <u>both</u> terminals upon command termination. Note that all error messages concerning access violations appear on t2.

Subroutine

Name: test_replace_list

The test_replace_list subroutine is called by the process_1_proc subroutine to test the mutual consistency of the hcs_$replace_acl and hcs_$list_acl subroutines. It calls the hcs_$replace_acl subroutine a series of times, each time attempting to replace the ACL of the segment replace_list. After each replacement attempt, it calls the hcs_$list_acl subroutine and compares the listed ACL with the ACL expected after the replacement attempt.

Usage

```
declare test_replace_list  entry (char(*), char(*), char(*)
                aligned, fixed bin(35));

call test_replace_list (u1, p1, path_name, code);
```

1) u1	is the Multics user name. See the writeup of the test_seg_acl command and the process_1_proc subroutine. (Input)
2) p1	is the Multics project id. (Input)
3) path_name	is the Multics path name for the temporary directory test_seg_acl_workspace_dir. (Input)
4) code	is a success code. See Notes below. (Output)

Notes

The value returned in code above is either zero or nonzero. If zero is returned, then no errors were encountered in the test of the mutual consistency of the hcs_$replace_acl and hcs_$list_acl subroutines. If nonzero, then some sort of error occurred and was noted on the stream "user_output".

Name: test_seg_acl

This command tests the basic access control mechanism of Multics by executing a series of tests to ascertain first, that the hcs_$append_branch, hcs_$add_acl_entries, hcs_$delete_acl_entries, hcs_$list_acl, and hcs_$replace_acl subroutines function correctly, and second that the ACL of a segment correctly controls the access of a process to that segment. The command must be issued twice, once from each of two terminals by different users. The two usages are distinguished by appearance of the arguments.

Usage (from first terminal)

```
test_seg_acl
```

Usage (from second terminal)

```
test_seg_acl u1 p1
```

1) u1	is the name of the user at the first terminal. See Notes below.
2) p1	is the name of the project of the user at the first terminal.

Notes

The test_seg_acl command is issued at the first terminal to begin the test of the basic access control mechanism. For later reference, this user´s process is referred to as p1 and his terminal as t1.

When test_seg_acl is called rom the first terminal, five main modules are referenced: test_append_list, test_add_list, test_delete_list, test_replace_list, and test_acl_use. The command creates the segments append_list_1, append_list_2, append_list_3, append_list_4, add_list, delete_list, replace_list, and try_me in a temporary directory in the process directory of P1. It creates also a temporary segment, test_seg_acl_mailbox, in the directory >udd>p1>u1. The following table lists the modules called, the temporary segments referenced, and the function of each module. This information is helpful in diagnosing any errors that might be reported.

Module	Using	Purpose
test_append_list	append_list_1 append_list_2 append_list_3 append_list_4	Test the mutual consistency of the subroutines hcs_$append_branch and hcs_$list_acl.
test_add_list	add_list	Test the mutual consistency of the subroutines hcs_$add_acl_entries and hcs_$list_acl.
test_delete_list	delete_list	Test the mutual consistency of the subroutines hcs_$delete_acl_entries and hcs_$list_acl.
test_replace_list	replace_list	Test the mutual consistency of the subroutines hcs_$replace_acl and hcs_$list_acl.
test_acl_use	try_me	Test that the ACL of try_me does in fact control the access of process P2 (See _Notes_ following) to try_me.

After calling test_seg_acl from the first terminal, instructions are printed telling the user to login at a second terminal under a different user and/or project id. This second user is referred to as "u2", his project id as "p2", his process as "P2", and the second terminal as "t2".

Process P2 receives instructions from P1 to access the segment, try_me. It returns the results of its access attempts to P1.

If the test_seg_acl command encounters any error in the basic access control mechanism, the error is noted on t1 and ready messages appear on both terminals. If no errors occur, then ready messages simply appear on both terminals upon command termination.

Errors

Certain errors print out on t1 in coded form. That basic form is:

```
          ts_acl:   -system_status_code-
                    Segment = "pathname"
                    Error number = "number"
                    -Optional information-
```

The following tables give the details for these coded error messages. A dash indicates the absence of optional information. It should be noted that the sequence of error messages in these tables correspond to the actual sequence of tests being made in a particular test module.

Segment	Error number	Optional information	Situation
append_list_i	1	-	Could not create segment giving P1 rw access.
	10	Expected ACL	Could not list ACL of segment.
	20	Listed ACL Expected ACL	ACL incorrectly listed.
add_list	1	-	Could not create segment giving P1 rw access.
	10	-	Could not create a project name different from p1.

Segment	Error number	Optional information	Situation
	20	-	Added to ACL: u1.p2.* r a.b.c.d rew
	30	-	No flag on: a.b.c.d rew
	40	Expected ACL	Could not list ACL of segment.
	50	Listed ACL Expected ACL	ACL incorrectly listed
	60	-	Could not add to ACL: u1.p2.* r
	70	Expected ACL	Could not list ACL of segment.
	80	Listed ACL Expected ACL	ACL incorrectly listed.
	90	-	Could not change ACL entry: u1.p2.* r to: u1.p2.* re
	100	Expected ACL	Could not list ACL of segment.
	110	Listed ACL Expected ACL	ACL incorrectly listed.
	120	-	Could not create a user name different from u1.

Segment	Error number	Optional information	Situation
	130	-	Could not add to ACL: u2.p2.* re
	140	Expected ACL	Could not list ACL of segment.
	150	Listed ACL Expected ACL	ACL incorrectly listed.
	160	-	Could not add to ACL: u2.p2.b rew
	170	Expected ACL	Could not list ACL of segment.
	180	Listed ACL Expected ACL	ACL incorrectly listed.
	190	-	Could not add to ACL: *.p1.* r u2.*.* r u1.p1.* rew *.*.* e
	200	Expected ACL	Could not list ACL of segment.
	210	Listed ACL Expected ACL	ACL incorrectly listed.
delete_list	1	-	Could not create segment, giving P1 rw-access.
	10	-	Could not create a user name different from u1.

Segment	Error number	Optional information	Situation
	20	-	Could not create a project name different from p1.
	30	-	Could not create a project name different from p1 and p2.
	40	-	Could not add to ACL: u1.p1.a rew u2.p2.a rew u2.p3.a re *.p2.* r
	50	-	Could not create a user name different from u1 and u2.
	60	-	Deleted from ACL: u1.p1.* u2.p2.a u3.p4.* *.SysDaemon.* a.b
	70	-	No flag on: a.b
	80	Expected ACL	Could not list ACL of segment.
	90	Listed ACL Expected ACL	ACL incorrectly listed.

Segment	Error number	Optional information	Situation
	100	-	Could not delete from ACL: u1.p1.* u2.p2.a u3.p4.* *.SysDaemon.*
	110	-	Flag on: u3.p4.*
	120	Expected ACL	Could not list ACL of segment.
	130	Listed ACL Expected ACL	ACL incorrectly listed.
replace_list	1	-	Could not create segment, giving P1 rw-access.
	10	-	Could not create a user name different from u1.
	20	-	Could not create a project name different from p1.
	30	-	Could not replace ACL with: u1.p1.a rew *.*.* r *.SysDaemon.* rw u2.p2.* rw
	40	Expected ACL	Could not list ACL of segment.

Segment	Error number	Optional information	Situation
	50	Listed ACL Expected ACL	ACL incorrectly listed.
	60	-	Could not create a user name different from u1 and u2.
	70	-	Replaced ACL with: u3.*.* r a.b.c.d rew
	80	-	No flag on: a.b.c.d rew
	90	Expected ACL	Could not list ACL of segment.
	100	Listed ACL Expected ACL	ACL incorrectly listed.
	110	-	Could not replace ACL with empty ACL.
	120	-	Could not list supposedly empty ACL of segment.
	130	Listed ACL	Listed ACL was not empty.
try_me	1	-	Could not create segment, giving P1 rew-access.
	10	-	Could not create a user name different from u1 and u2.

Segment	Error number	Optional information	Situation
	20	-	Could not create a project name different from p1 and p2.
	30	-	Could not create a tag value different from * and the tag value associated with u2 being logged onto t2.
	40	-	Could not add to ACL: u2.p2.x rew u2.p3.a rew u3.p2.a rew u2.p2.a null u2.p2.* rew u2.*.a rew u2.*.* rew *.p2.a rew *.p2.* rew *.*.a rew *.*.* rew
	50,90,130 170,210,260 310,360,410 460,510,560 710	-	Could not wake P2 to have it access segment.
	60,100,140 180,220,270 320,370,420 470,520,570 720	-	P1 could not go blocked.

Segment	Error number	Optional information	Situation
	70,110 150,190 230,280 330,380 430,480 530,580 730	Listed ACL The P2 access data (See following text and table.)	P2 reported improper access to segment.
	80	-	Could not change ACL entry: u2.p2.a null to: u2.p2.a r
	120	-	Could not change ACL entry: u2.p2.a r to: u2.p2.a re
	160	-	Could not change ACL entry: u2.p2.a re to: u2.p2.a rw
	200	-	Could not change ACL entry: u2.p2.a rw to: u2.p2.a rew
	240	-	Could not delete: u2.p2.a rew

Segment	Error number	Optional information	Situation
	250	-	Could not change: u2.p2.* rew to: u2.p2.* r
	290	-	Could not delete: u2.p2.*
	300	-	Could not change: u2.*.a rew to: u2.*.a r
	340	-	Could not delete: u2.*.a
	350	-	Could not change: u2.*.* rew to: u2.*.* r
	390	-	Could not delete: u2.*.*
	400	-	Could not change: *.p2.a rew to: *.p2.a r
	440	-	Could not delete: *.p2.a
	450	-	Could not change: *.p2.* rew to: *.p2.* r

Segment	Error number	Optional information	Situation
	490	-	Could not delete: *.p2.*
	500	-	Could not change: *.*.a rew to: *.*.a r
	540	-	Could not delete: *.*.a
	550	-	Could not change: *.*.* rew to: *.*.* r
	700	-	Could not replace ACL with: u2.p2.x rew u2.p3.a rew u3.p2.a rew u1.p1.a rew

When process P2 reports improper access to the segment try_me, then the optional information in the above error message is further coded as:

```
Error on other terminal = "number"
          Where status_code = "system_status_code"
                condition_found = "condition_name"
                word_read = "string"
                result_of_execution = "string"
                ptr_try_me = "ptr"
```

This information gives the reason for P2 reporting improper access to the segment try_me, which is the following ALM program:

```
" Word 0 is a constant that is used to check on read
"     access.
" Word 1 is open to check for write access.
" Word 2 is the entry point to check execute access.

c:   oct 252525252525         in word 0
     oct 0                    in word 1
     lda c                    in word 2
     sta ap|2,*
     short_return
     end
```

The following table details this optional information coding.

Error on other terminal	Meaning
1000	P2 was able to initiate the segment try_me, and perhaps read it. (See value in status_code, word_read, or ptr_try_me.)
2000	P2 was not able to read the segment try_me. (See value in status_code, condition_found, word_read, or ptr_try_me.)
2100	P2 did not encounter the condition "no_execute_permission" when attempting to execute try_me. (See value in status_code, condition_found, or result_of_execution.)
2200	P2 did not encounter the condition "no_write_permission" when attempting to write into try_me. (See value in status_code, or condition_found.)

Error on other terminal	Meaning
2300	P2 did meet "no_write_permission", but nevertheless damaged word 1 of try_me. (See value in word_read, which is the damaged contents of word 1 of try_me.)
3000	P2 was not able to read try_me. (See value in status_code, condition_found, or word_read.)
3100	P2 was not able to execute try_me. (See value in status_code, condition_found, or result_of_execution.)
3200	P2 did not encounter the condition "no_write_permission" when attempting to write into try_me. (See value in status_code, or condition_found.)
3300	P2 did meet "no_write_permission", but nevertheless damaged word 1 of try_me. (See value in word_read, which is the damaged contents of word 1 of try_me.)
4000	P2 was not able to read try_me. (See the value in status_code, condition_found, or word_read.)
4100	P2 was not able to write into try-me. (See value in status_code or condition_found.)

Error on other terminal	Meaning
4200	P2 was able to write into try_me, but did so incorrectly. (See value in word_read, which should have been all 7s after write.)
4300	P2 did not encounter the condition "no_execute_permission" when attempting to execute try_me. (See value in status_code, condition_found, or result_of_execution.)
5000	P2 was not able to read try_me. (See value in status_code, condition_found, or word_read.)
5100	P2 was not able to execute try_me. (See value in status_code, condition_found, or result_of_execution.)
5200	P2 was not able to write into try_me. (See value in status_code, or condition_found.)
5300	P2 was able to write into try_me, but did so incorrectly. (See value in word_read, which should have been all 7s after write.)

Error on other terminal	Meaning
6000	P2 did not encounter the condition "seg_fault_error" when attempting to read the previously initiated try_me after all access rights had been removed. (See value in status_code, condition_found, or word_read.)

Name: test_seg_auth, tsa

This command utilizes a special test directory to check that the access isolation mechanism works properly with respect to segments. In order to use this command properly, the special test directory must first be created using the exec_com "create_test_seg.ec".

Usage

```
     test_seg_auth -dirname-
```

1) dirname	is the pathname of the special test directory described below. If missing, the working directory is used.

Notes

The authorization of the process calling this command must be a certain level and category set combination as specified in the write-up to create_test_seg.ec. The user does not need system privilege to run this command -- in fact, he probably shouldn´t have it if the controls are to be tested properly.

If the test succeeds, no error messages will be printed. If the test fails, a message will be printed indicating the reason for the failure (bad status code or condition), the expected status code or condition, and the segment that was being referenced when the error occurred. The access class of the segment can be determined from the segment´s pathname (see the write-up of create_test_seg.ec).

Subroutine

Name: tipc_set_up

The tipc_set_up subroutine is called by the test_ipc command from the first terminal. It does the following:

1) creates a temporary segment "multi_process_info" in the home directory of the user at the first terminal.

2) converts the six arguments supplied to the test_ipc command to internal form. It then stores them in the segment multi_process_info.

3) prints messages instructing the user to go to a second terminal and call test_ipc without arguments.

4) accepts input from the user about his session on that second terminal.

5) does a new_proc to a process with authorization equal to auth1 as in the writeup of the test_ipc command.

Usage

```
   declare tipc_set_up  entry (char(*), char(*), char(*), char(*),
                 char(*), char(*));

   call tipc_set_up (str1, str2, str3, str4, str5, str6);
```

1) str1	is the authorization auth1. See the writeup of the test_ipc command. (Input)
2) str2	is the authorization auth2. (Input)
3) str3	is the authorization auth3. (Input)
4) str4	is the authorization auth4. (Input)
5) str5	is the authorization auth5. (Input)
6) str6	is the authorization auth6. (Input)

Subroutine

Name: try_dir_reference_

This subroutine references a given directory using all the hcs_ calls documented in the MPM (including SWG). The caller supplies the name of a directory, and the effective access mode he expects he has on that directory. This subroutine then checks to make sure that the expected access mode is enforced by all hcs_ calls that depend on that mode.

Usage

```
     declare try_dir_reference_ entry (char(*), char(*), char(*),
          char(*), bit(1), fixed bin(35));

     call try_dir_reference_ (parent, dirname, segname, mode, upgrade,
          error);
```

1) parent	is the name of the directory to which access is to be tested. (Input)
2) dirname	is the name of a subdirectory within parent. (Input)
3) segname	is the name of a segment within parent. (Input)
4) mode	is the expected effective access mode to parent. This value may be one of the strings: "", "n", "s", or "sm". (Input)
5) upgrade	is "1"b if the access class of parent is not less than the current process authorization. In this case, the parent of parent must be at an equal or lower access class than the current process authorization. If parent is at an equal or lower access class, this value must be "0"b. (Input)
6) error	is zero if no errors or inconsistencies occurred during the test. If nonzero, a positive value is a standard storage system status code indicating that the pathname of parent was bad, or that some temporary segments could not be created. If -2, the test was completed, but some error was detected in the system. The error

message(s) is printed on user_output. (Output)

Notes

There are certain restrictions on the contents of parent and its attributes. They are listed below:

	parent	dirname	segname
quota	>1	0	--
ACL for user	(see below)	sma	rew
bitcount	--	0	1
rings	7,7	7,7	4,4,4
safety switch	off	off	off
max length	>1	>1	1024 words

The ACL of parent depends on the mode and properties to be tested. If only ACLs are being tested, as opposed to access isolation, the access mode to parent should be the mode being tested. If access isolation is being tested, the ACL of parent should be sma for the user. The effective mode, in this case (which depends on the access class of parent or its parent), should be the mode being tested. Note that if upgrade is set, the effective mode should be "null", since there is never any access to a directory of a higher access class.

In addition to the above attributes, the segment should contain all zeros except the first bit, which should be "1"b. The ~~directory~~ dirname should be empty, and parent should contain no other entries except dirname and segname.

Subroutine

Name: try_reference_

This subroutine attempts to reference a specified segment in one of several modes (read, write, execute, or call) and returns any condition name or error code resulting from the reference.

Entry: try_reference_$seg

This entry requires a pointer to the segment to be referenced.

Usage

```
declare try_reference_$seg entry (ptr, char(1), bit(36) aligned,
     char(*), char(32), fixed bin(35));

call try_reference_$seg (segptr, mode, data, condition_wanted,
     condition_name, code);
```

1) segptr	is a pointer to the segment and word to be referenced. (Input)
2) mode	is one of the following: "r" read the specified word. "w" write the specified word. "e" call the specified word using a transfer instruction. "c" call the specified word using a call6 or callsp instruction. (Input)
3) data	If "r" was specified, the data read will be stored here. (Output) If "w" was specified, this is the data to be written. (Input) If "e" or "c" was specified, this argument will be passed to the procedure being referenced. The procedure may store a value into this argument or it may obtain a value. (Input/Output)

4) condition_wanted	If not zero length, this should be a condition name, such as "no_read_permission", which will be interpreted as a condition to be expected by the particular reference. If the condition resulting is the expected condition, no condition name will be returned in "condition_name". If no condition occurred, and "condition_wanted" is not null, the string "access_allowed" will be returned. If this argument is zero length or blank, any condition that occurs will be returned. (Input)
5) condition_name	If the condition resulting from the reference does not match "condition_wanted", the condition name is returned here. If "condition_wanted" was not null, and no condition occurred, the string "access allowed" will be returned here. (Output)
6) code	This is normally zero for most hardware conditions. However, if a call to find_condition_info_ supplies a valid error_table_ code, that code will be returned here. If this occurs, the condition mechanism has probably malfunctioned.

Entry: try_reference_$file

This entry operates similar to try_reference_$seg, except that the name of the segment is supplied instead of a pointer. The main difference is that the status code may reflect a failure to initiate the segment due to null access or bad pathname. If initiate fails, the status code should always be non zero and condition_name will always be null.

Usage

```
declare try_reference_$file entry (char(*), char(*), ptr, fixed
     bin, char(1), bit(36) aligned, char(*), char(32), fixed
     bin(35));

call try_reference_$file (dirname, ename, ptr, offset, mode,
     data, condition_wanted, condition_name, code);
```

1) dirname is the directory name portion of the pathname of the segment. If zero length, the working directory will be used. (Input)

2) ename is the name of the segment. (Input)

3) segptr is a pointer to the word referenced. It is null if initiate failed. (Output)

4) offset is the location within the segment to be referenced. (Input)

5) mode is as above. (Input)

6) data is as above. (Input/Output)

7) condition_wanted
is as above. (Input)

8) condition_name
is blank if code is nonzero. Otherwise, it is set as above. (Output)

9) code If not zero, initiation failed for some reason or the condition mechanism failed as described above. If zero, initiation was successful. (Output)

Entry: try_reference_$entry

This entry accepts a pathname of the segment and an entry name of the form pathname$entryname. The search rules are used to locate the segment if the pathname is just a segment name in a manner similar to the search for a command. If "$entryname" is not specified, it is assumed to be the same as the segment name. This entry may return a status code if the segment could not be found or initiated.

Usage

```
declare try_reference_$entry entry (char(*), char(1), bit(36)
     aligned, char(*), char(32), fixed bin(35));
```

```
call try_reference_$entry (pathname, mode, data,
     condition_wanted, condition_name, code);
```

1) pathname is the relative pathname of the segment as described above. The specific entry point specified will be the word referenced. (Input)

2) - 6) are as above.

APPENDIX IV

LISTINGS

```
&goto &ec_name
& ********************** CREATE_TEST_SEG *****************

& This exec_com creates a directory with subdirectories and segments required for
& the test_seg_auth command.  It is called as follows:
&      exec_com create_test_seg path -acc class1 class2 class3 class4 class5 class6
& All arguments are required.  "path" is the name of the directory to be created.

& Second argument may be -acc, which is followed by 6 arguments representing the access classes of
& the five directories in the following order:
&           1. lower level, equal category set:
&           2. higher level, equal category set
&           3. equal level and category set
&           4. equal level and subset of category
&           5. equal level and superset of category
&           6. equal level and isolated category set

&label create_test_seg
&command_line off
set_com_line 500
&if [exists segment test_seg_auth_]
&then &goto create_test_dir
&print &ec_name: Segment test_seg_auth_ was not found in the working directory.
&quit

```

```
& ********************** CREATE_TEST_DIR ****************

& This exec_com creates a directory with subdirectories required for the test_dir_auth
& command.  It is called as follows:
&          exec_com create_test_dir path -acc class1 class2 class3 class4 class5 class6
& where "path" is the pathname of the test directory.  If a directory by that name already exists,
& the user will be asked whether he wants to delete it.  The parent of "path" must exist and
& must have a quota of at least 25.  The user must have modify permission to the parent.
& The -acc and following arguments are explained above.

&label create_test_dir
&command_line off
set_com_line 500
&if [not [equal "&1" ""]]
&then &goto ctd0
&print &ec_name: Expected argument missing.
&print Usage is: exec_com &ec_name path -acc class1 ... class6
&quit

&label ctd0
&if [equal "&2" -acc]
&then &goto ctd3
&print &ec_name: "-acc" must be the second argument, followed by access classes.
&quit

&label ctd3
&if [and [not [equal "&8" ""]] [equal "&9" ""]]
&then &goto ctd4
&print &ec_name: There must be exactly 6 arguments following -acc.
&goto quit

&label ctd4
&if [equal [string [user auth]] [string [short_string system_low]]]
&then &goto we_are_low
&print &ec_name: You must be at system_low to execute this command.
&quit

&label we_are_low
&if [exists directory [directory &1]]
&then &goto ctd1
&print &ec_name: Parent directory of path specified does not exist. &1
&quit

&label ctd1
assoc_set working_dir [wd]
change_wdir [directory &1]
&if [not [exists directory &1]]
&then &goto ctd2
&if [not [query "&ec_name: Directory &1 already exists.  Do you want to delete it?"]]
&then &goto quit
answer yes -bf delete_dir &1
&if [equal "&2" ""]
&then &goto ctd8

&label ctd8
&if [exists directory &1]
&then &go to quit
```

```

&label ctd2
&if [equal &ec_name create_test_seg]
&then assoc_set min_quota 24
&else assoc_set min_quota 30
&if [equal &ec_name create_test_seg]
&then &goto ctd6
&if [not [equal &ec_name create_test_auth]]
&then &goto ctd12

& If create_test_auth is called, we need to count the number of arguments, which becomes the
& number of directories that must be created (plus one for system_low).

assoc_set min_quota 1
do "assoc_set min_quota [plus [assoc min_quota] 1]" &2
assoc_set min_quota [times [assoc min_quota] 2]

&label ctd12
&if [ngreater [minus [quota [directory &1]] [quota_used [directory &1]]] [assoc min_quota]]
&then &goto ctd6
ioa_ "&ec_name: There must be at least "[plus [assoc min_quota] 1]" records of quota left in parent directory. &1"

&label quit
change_wdir [assoc working_dir]
assoc_set (working_dir dirs levels short_names min_quota) ""
&quit

&label ctd6
create_dir &1
&if [not [exists directory &1]]
&then &goto quit
&if [exists segment [home_dir]>create_test_acl] &then set_acl &1 sma ([all [home_dir]>create_test_acl])
move_quota &1 [assoc min_quota]
&if [not [equal [quota &1] [assoc min_quota]]]
&then &go to quit
change_wdir &1
&if [equal &ec_name create_test_auth]
&then &goto ctd0
assoc_set dirs "(lower_equal higher_equal equal_equal equal_subset equal_superset equal_isolated)"
&if [equal &ec_name create_test_seg]
&then create_dir [assoc dirs] -access_class (&3 &4 &5 &6 &7 &8) -quota 3
&else create_dir [assoc dirs] -access_class (&3 &4 &5 &6 &7 &8) -quota 4
&if [ngreater [index [exists directory [assoc dirs]] false] 0] &then &goto quit
&if [exists segment [home_dir]>create_test_acl]
&then do "set_acl &(1) sma ([all [home_dir]>create_test_acl])" [assoc dirs]

& We've created the upgraded directories.  Now go to the home directory and
& set up the segments necessary to store information so that we can new_proc
& to the various levels and set up stuff in these upgraded directories.

& The segment "who" contains the name of this exec_com.

assoc_set first_auth &3

&label common_end
change_wdir [home_dir]
do "if isfile &(1) -then ""truncate &(1)""" (who ac_names original_wdir pathname)
file_output who; ioa_ &ec_name; console_output
```

```
&if [equal &ec_name create_test_authn]
&then file_output ac_names; ioa_ &[do "[short_string &(1)]&&(2)&" &2 [assoc short_names]]; console_output
&else file_output ac_names; ioa_ &[do "[short_string &(1)]&&(2)&" (&3 &4 &5 &6 &7 &8) [assoc dirs]]; console_output
file_output pathname; ioa_ &1; console_output
file_output original_wdir; ioa_ [assoc working_dir]; console_output
new_proc -authorization [assoc first_authn]
& we should not get past here.
ioa_ "&ec_name: new_proc to authorization """"[assoc first_authn]"""" failed."
&quit
```

```
& **************** CREATE_TEST_AUTH.EC **********

& Create the directory for authorization_tester.
& Called with two arguments:
&    exec_com create_test_auth path "(class1 class2 ... classn)"

& path      is the pathname of the directory to be created.

& classi    are the names of all levels and all categories within system_high,
&           separated by spaces, enclosed in parentheses and quotes.  They may
&           be in any order. System low should not be used.

&label create_test_auth
&command_line off
set_com_line 500
&if [not [equal "&2" ""]]
&then &goto cta1

&label cta3
&print &ec_name: Second argument must be a quoted parenthesized list of access classes.
&quit

&label cta1
&if [equal "&3" ""]
&then &goto cta2
&print &ec_name: Too many arguments. &3
&quit

& Set first_last equal to a quoted string containing the first and last characters of &2.
& Since these are expected to be parentheses, we have to go through the mishmosh below.

&label cta2
assoc_set first_last [substr """" 1][substr "&2" 1 1][substr "&2" [length "&2"] 1][substr """" 1]
assoc_set first_last [substr """""""" 1][assoc first_last][substr """""""" 1]
&if [not [equal [assoc first_last] "()"]]
&then &goto cta3
&goto cta4

& Come back here when path has been created, and sufficient quota has been moved to it.
& Set the variable encoded names to "(e1 e2 ... en)", where ei is the encoded
& form of classi.

&label cta0
assoc_set short_names [string "(" [do "[short_string &r1]""_auth """ &2] ")"]
&if [nequal [index [assoc short_names] "*"] 0]
&then &goto cta4
&print &ec_name: Some access class in "&2" is invalid.
&goto quit

&label cta4
create_dir [assoc short_names] -access_class &2 -quota 1
&if [ngreater [index [exists directory [assoc short_names]] false] 0] &then &goto quit
&if [exists segment [home_dir]>create_test_acl]
&then do "set_acl &(1) sma ([all [home_dir]>create_test_acl])" [assoc short_names]
assoc_set first_auth ""
do "if -not arg [assoc first_auth] -then ""assoc_set first_auth &(1)""" &2
create_dir [short_string system_low]_auth
```

```
create [short_string system_low]_auth>seg
&if [exists segment [home_dir]>create_test_acl] &then set_acl [short_string system_low]_auth sma ([all [home_dir]>create_test_acl]
  ); set_acl [short_string system_low]_auth>seg r ([all [home_dir]>create_test_acl])
&goto common_end
```

```
& *************************** CREATE_TEST_START_UP ******************************

& This exec_com creates the interior directories and segments for the
& create_test_dir, create_test_seg, and create_test_auth exec_coms.
& It should be called by the user's start_up.ec after executing
& one of the above three exec_coms.  In the home directory there is assumed
& to be several segments that have been created by the original exec_com
& that is used to drive this exec_com.  The original exec_com was called from
& system_low, and thus the data bases created can be read by this exec_com at each new
& authorization in order to determine what to do next.

& The segments in the home directory contain one line with the following information:

& who         contains the name or the original exec_com (e.g. "create_test_dir") so
&             that this exec_com knows from where it originated.  If this segment is
&             not found, this exec_com merely returns, doing nothing.  Thus, it
&             is safe to place a call to this exec_com at the end of the start_up.ec
&             in the case of a new_proc, and it will only be called when the segment
&             "who" exists.

& ac_names  contains the following string:

&                 $acname1$dirname1$acname2$dirname2$...$acnamen$dirnamen$

&             where  acnamei  is the short name of an authorization, one of which
&                             corresponds to the current authorization.
&                    dirnamei is the name of the directory that has previously been
&                             created and upgraded to acnamei.

& original_wdir contains the pathname of the original working directory when the original exec_com
&             was called by the user.

& pathname  contains the pathname of the parent directory that contains the dirnamei's.

& create_test_acl is optionally created by the user.  If it exists, it is assumed
&             to contain a list of group identifiers (one per line) that are to be
&             put on the ACLs of all the directories and segments created by this exec_com.
&             The access modes are determined within this exec_com.

&label create_test_start_up
&if [not [exists segment [home_dir]>who]]
&then &quit
&command_line off
&input_line off
change_wdir [home_dir]
&if [equal [all who] create_test_dir]
&then &goto ctsu0
&if [equal [all who] create_test_seg]
&then &goto ctsu0
&if [equal [all who] mbx_test] &then &goto ctsu0
&if [equal [all who] create_test_auth]
&then &goto ctsu0
ioa_ "&ec_name: The name """[all who]""" as specified in the segment"
ioa_ "[home_dir]>who is not a name of an exec_com for test procedures."
&quit

&label ctsu0
```

```
set_com_line 1000

& if the current authorization is sytem_low, we must be done.

&if [equal [user auth] [short_string system_low]]
&then &goto clean_up

& find current authorization in ac_names segment.

assoc_set pos [index [all ac_names] &[string [user auth]]&]
&if [not [nequal [assoc pos] 0]] &then &goto cur_auth_found
ioa_ "&ec_name: The current process authorization of """[user auth]""" was not found"
ioa_ "in the segment [home_dir]>ac_names."
&goto clean_up

& now get whatever text follows it in ac_names.

&label cur_auth_found
assoc_set pos [plus [assoc pos] [length &[string [user auth]]&]]
assoc_set len [index [substr [all ac_names] [assoc pos]] $]
assoc_set text [substr [all ac_names] [assoc pos] [minus [assoc len] 1]]
&if [equal [all who] mbx_test] &then &goto mbx_return

& For create_test_....ec, this text is a directory name.

change_wdir [all pathname]>[assoc text]
&if [equal [all [home_dir]>who] create_test_auth]
&then &goto ctsu1

& for create_test_dir and create_test_seg, create the subdirectory called "dir"

create_dir dir
&if [exists segment [home_dir]>create_test_acl] &then set_acl dir sma ([all [home_dir]>create_test_acl])
&if [equal [all [home_dir]>who] create_test_seg]
&then &goto ctsu2

& for create_test_dir, the segment "seg" has some things put into it and its branch

create seg
set_bit_count seg 1
set_max_length seg 1024
&attach
debug
.mb
/seg/=400000000000
.q
assoc_set modes rew

& All done at current process level.  Now we have to do a new_proc to
& the next name in ac_names.
& If we used up all the names, we go to system_low.

&label next_new_proc
&if [exists segment [home_dir]>create_test_acl] &then set_acl seg [assoc modes] ([all [home_dir]>create_test_acl])

&label next_new_proc_1
change_wdir [home_dir]
assoc_set pos [plus [assoc pos] [assoc len]]
```

```
&if [ngreater [assoc pos] [length [all ac_names]]]
&then new_proc -authorization system_low
assoc_set len [index [substr [all ac_names] [assoc pos]] $]
new_proc -authorization [substr [all ac_names] [assoc pos] [minus [assoc len] 1]]
ioa_ "create_test_start_up.ec: new_proc to authorization """[substr [all ac_names] [assoc pos] [minus [assoc len]]]""" failed."
&print Returning to command level.
print_proc_auth

& Create segment with a program in it for create_test_seg

&label ctsu2
change_wdir dir
copy [all [home_dir]>original_wdir]>test_seg_auth_ seg
assoc_set modes rew
set_acl seg rew
&goto next_new_proc

& Create segment in upgraded directory for create_test_auth

&label ctsu1
create seg
set_acl seg r
assoc_set modes r
&goto next_new_proc

& On system_low, delete the temporary segments, restore the original
& working_dir, and make it look as if nothing happened.

&label clean_up
do "if is &(1) -then ""delete &(1)""" [who ac_names pathname]
&if [not [exists segment original_wdir]] &then &quit
change_wdir [all original_wdir]
delete [home_dir]>original_wdir
&quit

```

```
& ********************** MBX_TEST_START_UP **********************

& This exec_com sends messages of various authorizations to the mailbox of the
& user.  It is called by the user's start_up.ec after each new_proc after
& mbx_test.ec was called while at system_low.  This exec_com operates in a
& manner very similar to create_test_start_up.ec, and borrows much of
& its code.

&label mbx_test_start_up.ec
&goto create_test_start_up

& return here after the current authorization has been located in the
& ac_names segment and the variable text has been set to a letter
& A B C D E or F, indicating which of six authorizations we are at.

&label mbx_return
&command_line off

& If we're at level A, and we already have mail in the mailbox, then
& we've finished sending messages and now should try to read them.

&if [not [and [equal [assoc text] A] [have_mail]]] &then &goto send_mail
&print
&print mbx_test: Messages A, B, and C should follow, plus "Incorrect access" messages from mail regarding 2 and 3:
&print
answer yes -bf mail
&print
&print mbx_test: Messages B and C should now follow:
&print
answer no -bf mail

& Make sure that the mailbox can't be deleted.

&print
&print mbx_test: One final error message from mbx_delete:
&print
mbx_delete >udd>[user project]>[user name]>[user name]
&if [not [query "mbx_test: Everything as expected?"]]
&then &print mbx_test: Test failed.
new_proc -auth system_low

& Come here to send a message
& We must call this exec_com again, so we can easily pass arguments for insertion into the message.

&label send_mail
exec_com [directory &0]>mbx_test_part_2 [assoc text] [user auth]

& When done, do another new_proc.

&goto next_new_proc_1

& Enter here on second call, to mbx_test_part_2.

&label mbx_test_part_2
&attach
&command_line off
&input_line off
```

```
mail * [user name] [user project]
&1. This message is &2.
.
&detach
&quit

```

```
& **************************** MBX_TEST.EC ************************

& This exec_com makes security checks of mailbox controls.
& It is called with six arguments which are access classes as specified for
& create_test_seg.ec and create_test_dir.ec.

&label mbx_test
&command_line off
set_com_line 500
&if [not [equal "&1" ""]] &then &goto mbx0
&print &ec_name: Expected argument missing.

&label mbx_usage
&print Usage is: exec_com mbx_test class1 ... class6
&quit

&label mbx0
&if [and [not [equal "&6" ""]] [equal "&7" ""]] &then &goto mbx1
&print &ec_name: There must be exactly six arguments.
&goto mbx_usage

& Validate the six access class names. "short_string" returns "**" for invalid access class.

&label mbx1
&if [nequal [index [do "[short_string &(1)]" (&1 &2 &3 &4 &5 &6)] **] 0] &then &goto mbx2
&print &ec_name: Some access class specified is illegal.
&quit

&label mbx2
&if [equal [short_string system_low] [user auth]] &then &goto mbx3
&print &ec_name: You must be at system_low to execute this command.
&quit

& Now it's time to initialize the temporary segments in the home directory,
& as is done for the create_test_... exec_coms.

&label mbx3
&if [not [exists segment >udd>[user project]>[user name]>[user name].mbx]] &then &goto mbx4
&if [have_mail] &then &goto mbx5
&if [not [query "Do you want to delete your old mailbox?"]] &then &quit
mbx_delete >udd>[user project]>[user name]>[user name].mbx
&if [exists segment >udd>[user project]>[user name]>[user name].mbx] &then quit
&else &goto mbx4

&label mbx5
&print &ec_name: You have mail.  Get rid of it and call this exec_com again.
&quit

&label mbx4
mbx_create >udd>[user project]>[user name]>[user name]
&if [exists segment >udd>[user project]>[user name]>[user name].mbx] &then &goto mbx7
&print &ec_name: Couldn't create a new mailbox.  Test not run.
&quit

&label mbx7
do "if isfile [home_dir]>&(1) -then ""truncate [home_dir]>&(1)""" (who original_wdir ac_names)
file_output [home_dir]>ac_names; ioa_ $[do "[short_string &(1)]&&(2)&" (&3 &1 &4 &2 &5 &6 &3) (A B C D E F G)]; console_output
file_output [home_dir]>who; ioa_ &ec_name; console_output
file_output [home_dir]>original_wdir; ioa_ [wd]; console_output
&print Please ignore the next six "Input:" lines.
new_proc -authorization "&3"
& we should not get here.
&print &ec_name: new_proc to authorization "&1" failed.
&quit

```

```
**************************** AUDIT.EC ********************

& Exec_com to test auditing.  It assumes all the audit bits are set for this process.
& It tests each audit bit once by invoking the action that causes that condition to be
& detected.

&label audit
&command_line off
&if [not [equal [user auth] [short_string system_low]]] &then &goto auth_ok
&print &ec_name: You must not be at system_low when performing this test.
&quit

&label auth_ok
&if [exists directory &1] &then &goto dir_good
&print &ec_name: Directory not found. &1
&quit

&label dir_good
&if [exists segment &2] &then &goto seg_good
&print: &ec_name: Segment not found. &2
&quit

&label seg_good

assoc_set start_time [date_time]

& AUD-1: dir_init

list -pn &1

& AUD-2: seg_init

initiate &2

& AUD-3: mc_seg_init

mail

& AUD-4: no_access

audit no_access

& AUD-5: ipr_fault

audit ipr_fault

& AUD-6: acv_mode

audit acv_mode

& AUD-7: acv_ring

audit acv_ring

& AUD-8: no_wakeup

send_message [response "Enter name of any user logged in below "[user auth]": "] [response "Enter his project:"] Hello
```

```

& AUD-9: sys_priv enable

set_system_priv dir
set_system_priv ^dir

& AUD-10: ssa_ops

reclassify_dir &1 [user auth]

& AUD-11: no_attach

& AUD-12: no_mount

& AUD-13: mseg

set_max_length [home_dir]>[user name].mbx 1024
mail &2 [user name] [user project]
mail &2 [user name] [user project]
set_max_length [home_dir]>[user name].mbx 131072

& Now print the syserr_log for the user.  Print all audit entries since beginning of tests.

print_syserr_log -class 24 -from [string [assoc start_time]]
```

```
                    audit.pl1                              07/28/75  1529.8 ect Mon                              page 1

audit: proc;
dcl com_err_ entry options (variable);
dcl cu_$stack_frame_ptr entry (ptr);
dcl (not_in_read_bracket, illegal_procedure, no_write_permission) condition;
dcl cu_$arg_ptr entry (fixed bin, ptr, fixed bin, fixed bin(35));
dcl get_group_id_ entry returns (char(32) aligned);
dcl get_pdir_ entry returns (char(168) aligned);
dcl hcs_$append_branchx entry (char(*), char(*), fixed bin(5), (3) fixed bin(3), char(*),
          fixed bin(1), fixed bin(1), fixed bin(24), fixed bin(35));
dcl hcs_$initiate entry (char(*), char(*), char(*), fixed bin(1), fixed bin(2), ptr, fixed bin(35));
dcl hcs_$make_seg entry (char(*), char(*), char(*), fixed bin(5), ptr, fixed bin(35));
dcl hcs_$status_ entry (char(*), char(*), fixed bin(1), ptr, ptr, fixed bin(35));
dcl ioa_$ioa_stream_nnl entry options (variable);
dcl arg char(arglen) based(argptr);
dcl arglen fixed bin;
dcl argptr ptr;
dcl code fixed bin(35);
dcl error_table_$badopt external fixed bin(35);
dcl error_table_$namedup external fixed bin(35);
dcl error_table_$no_info external fixed bin(35);
dcl null builtin;
dcl 1 label_pair,
      2 ptr1 ptr,
      2 ptr2 ptr;
dcl privileged_instruction aligned bit(36) static init (
    "000000000000000000000101111000000000"b); /*SSCP instruction */
dcl label label based (addr (label_pair));
dcl arguments (6) char(16) static init (
    "no_access", "ipr_fault", "acv_mode", "acv_ring", "no_mount", "no_attach");
dcl segptr ptr;
dcl word bit(1) based (segptr) aligned;
dcl bit bit(1) aligned;
dcl i fixed bin;

call cu_$arg_ptr (1, argptr, arglen, code);
if code ^= 0 then do;
  call com_err_ (code, "audit", "Allowed arguments are:");
    do i = 1 to hbound(arguments,1)-1;
      call ioa_$ioa_stream_nnl ("error_output", "^a, ", arguments(i));
    end;
    call ioa_$ioa_stream_nnl ("error_output", "^a^/", arguments(hbound(arguments,1)));
    return;
    end;
do i = 1 to hbound(arguments,1);
  if arg = arguments(i) then goto x(i);
end;
call com_err_ (error_table_$badopt, "audit", arg);
return;

/* AUD-4: no_access */

x(1):call hcs_$append_branchx (get_pdir_(), "audit_dir", 3b, 7, get_group_id_(), 1, 0, 0, code);
      if code ^= 0 & code ^= error_table_$namedup then call coderr ("[pd]>audit_dir");
      call hcs_$status_ (before (get_pdir_(), " ") || ">audit_dir", "xx", 0, null(), null(), code);
      if code ^= error_table_$no_info then call coderr ("[pd]>audit_dir");
      return;

/* AUD-5: ipr_fault */
```

```

x(2):on illegal_procedure goto continue_ipr;
      label_pair.ptr1 = addr(privileged_instruction);
      call cu_$stack_frame_ptr (label_pair.ptr2);
      goto label;
continue_ipr: return;

/* AUD-6: acv_mode */

x(3):call hcs_$make_seg ("", "audit_seg", "", 01000b, segptr, code);
      if segptr = null then call coderr ("[pd]>audit_seg");
      on no_write_permission goto continue_acv_mode;
      word = "1"b;
      call no_fault ("no_write_permission (referencing [pd]>audit_seg)");
continue_acv_mode: return;

/* AUD-7: acv_ring */

x(4):call hcs_$initiate (">system_library_1", "hcs_", "", 0, 0, segptr, code);
      if segptr = null then call coderr (">system_library_1>hcs_");
      on not_in_read_bracket goto continue_acv_ring;
      bit = word;
      call no_fault ("not_in_read_bracket (referencing >system_library_1>hcs_)");
continue_acv_ring: return;

/* AUD-11: no_attach */

x(5):call not_implemented;

/* AUD-12: no_mount */

x(6):call not_implemented;


not_implemented: proc;
      call com_err_ (0, "audit", "Test not implemented. ^a", arguments(i));
      goto return;
end;

coderr: proc (message);
      dcl message char(*);
      call com_err_ (code, "audit", "This condition should not have occurred,
referencing ^a", message);
      goto return;
end;

no_fault: proc (message);
      dcl message char(*);
      call com_err_ (0, "audit", "The expected condition ^a was not raised.");
      return;
end;

return:

end;
```

```
authorization_tester: proc;

 dcl dummy fixed bin;
 dcl arglen fixed bin;
 dcl argno fixed bin;
 dcl code fixed bin(35);
 dcl dirname_length fixed bin;
 dcl seg fixed bin based(segptr);
 dcl category_number fixed bin;
 dcl return_length fixed bin;

 dcl error_table_$moderr external fixed bin(35);
 dcl error_table_$incorrect_access external fixed bin(35);
 dcl error_table_$badopt external fixed bin(35);

 dcl convert_authorization_$from_string entry (bit (72) aligned, char(*), fixed bin(35));
 dcl convert_authorization_$to_string_short entry (bit(72) aligned, char(*), fixed bin(35));
 dcl convert_authorization_$to_string entry (bit(72) aligned, char(*), fixed bin(35));
 dcl cv_dec_check_ entry (char(*), fixed bin(35)) returns (fixed bin(35));
 dcl hcs_$get_authorization entry (bit(72) aligned, bit(72) aligned);
 dcl com_err_ entry options(variable);
 dcl sw bit(1) init("1"); /* determines what kind of error message to print */
 dcl (ioa_, ioa_$nnl, ioa_$rs) entry options(variable);
 dcl bit_to_integer_ entry (bit(*)) returns (char(*));
 dcl get_dir_arg_ entry (fixed bin, char(*), fixed bin(35));
 dcl cu_$arg_ptr entry (fixed bin, ptr, fixed bin, fixed bin(35));
 dcl cu_$arg_count entry (fixed bin);
 dcl expand_path_ entry (ptr, fixed bin, ptr, ptr, fixed bin(35));
 dcl convert_status_code_ entry (fixed bin(35), char(8) aligned) returns(char(100) aligned);
 dcl get_wdir_ entry returns (char(168) aligned);
 dcl try_reference_$file  entry (char(*), char(*), ptr, fixed bin, char(1), bit(36) aligned,
         char(*), char(32), fixed bin(35));

 dcl argptr ptr;
 dcl segptr ptr;

 dcl arg char(arglen) based (argptr);
 dcl return_string char(300);
 dcl current_string char(128);
 dcl dirname char(168) init (get_wdir_());
 dcl ename char(32) aligned;
 dcl chars char(32);
 dcl condition char(32);

 dcl category bit(36);
 dcl level_found bit(1) init ("0"b);
 dcl command bit(1);

 dcl null builtin;

 dcl 1 real aligned,
       2 category bit(36) unaligned init (""b),
       2 level fixed bin(17) unaligned,
       2 pad bit(18) unaligned;

 dcl 1 (current, working_class, max_level, system_high) aligned like real;

 dcl real_bits bit(72) aligned based (addr(real));
dcl current_bits bit(72) aligned based (addr(current));
dcl working_class_bits bit(72) aligned based (addr(working_class));
dcl system_high_bits bit(72) aligned based (addr(system_high));
dcl max_bits bit(72) aligned;

dcl (working_string, last_working_string) char(150);

/* entry from command call */

command = "1"b;

system_high_bits = ""b;
call cu_$arg_count (argno);

do argno = 1 to argno;
  call cu_$arg_ptr (argno, argptr, arglen, code);
  if arg = "-max" then do;
    call cu_$arg_ptr (argno+1, argptr, arglen, code);
    call convert_authorization_$from_string (system_high_bits, arg, code);
    if code ^= 0 then do;
argerr:   call com_err_ (code, "authorization_tester", arg);
          return;
          end;
    argno = argno + 1;
    end;
  else do;
    if substr (arg, 1, 1) = "-" then do;
          code = error_table_$badopt;
          goto argerr;
          end;
    call get_dir_arg_ (argno, dirname, code);
    if code ^= 0 then do;
      call com_err_ (code, "authorization_tester", dirname);
      return;
      end;
    end;
end;
goto common;

/* entry from subroutine call */

authorization_tester_: entry (maxauth, argument, message);
 dcl maxauth bit(72) aligned;
 dcl argument char(*);          /* name of directory or working directory */
 dcl message char(*);           /* error message is put here,if any */
 if argument = "" then dirname = get_wdir_();
                  else dirname = argument;
 system_high_bits = maxauth;
 command = "0"b;

common:

dirname_length = index (dirname, " ") - 1;
substr (dirname, dirname_length+1, 1) = ">";
if system_high_bits = ""b
then call convert_authorization_$from_string (system_high_bits, "system_high", code);

/*
```

```
*/
/* This section tries to determine the authorization level of the
   process by referencing segments of different levels in the
   directory dirname.  Each of the directories has a name which is the short
   version of the access class of the directory with the suffix "_auth".  Within each directory is
   a zero length segment with the name "seg".  This section of the program
   starts at level 0 and references up until it gets to a segment it can't
   read.  The last segment readable must be the current level.  It also
   makes sure that segments above that level can't be initiated.
*/

working_class.category, working_class.pad = ""b;

do working_class.level = 0 to system_high.level;
   call convert_authorization_$to_string_short (working_class_bits, chars, code);
   chars = before (chars, " ") || "_auth";
   dirname = substr (dirname, 1, dirname_length+1) || chars;
   call convert_authorization_$to_string (working_class_bits, working_string, code);

   /* try to read or initiate segment */

   call try_reference_$file (dirname, "seg", null, 0, "r", "0"b, "", condition, code);
   if code = error_table_$incorrect_access   /* this is expected if level has been passed */
   then level_found = "1"b;         /* set flag that level probably was found */
   else
      if code = 0                   /* no code, initiate was allowed */
      then
         if level_found             /* was level found already? */
         then do;                   /* if so something is wrong */
            call ioa_$rs ("Initiate allowed on ^a segment but not on ^a segment",
                        return_string, return_length, last_working_string, working_string);
            goto error_return;
            end;
         else do;                   /* level not already found  */
            if condition = ""      /* any condition code?      */
            then real_bits = working_class_bits; /* no, this is <= real level -- save value */
            else do;
               call ioa_$rs ("Condition ""^a"" signalled when^/reading ^a segment.  No condition expected.",
                           return_string, return_length, condition, working_string);
               goto error_return;
               end;
            end;
      else                          /* initiate not allowed, but not expected code */
         if level_found             /* What we expected depends on whether level was found */
         then do;
              call ioa_$rs ("Status code ""^a"" returned on initiate
of ^a segment instead of ""^a""", return_string, return_length,
                      convert_status_code_ (code, ""), working_string, convert_status_code_ (error_table_$incorrect_access, ""));
              goto error_return;
              end;
         else do;
              call ioa_$rs ("Status code ""^a"" returned on initiate
of ^a segment instead of ""^a"" or none.", return_string, return_length,
                      convert_status_code_ (code, ""), working_string, convert_status_code_ (error_table_$incorrect_access, ""));
              goto error_return;
              end;
   last_working_string = working_string;

```

```
end;
/*
```

```
*/
/* Now that we have the level, get the category in a similar manner */

/* We cycle through all 18 categories, but only perform the test for those
   categories included within system_high (or the max as specified in the
   call to the command. */

working_class.level = 0;
do category_number = 1 to 18;
 if substr (system_high.category, category_number, 1) then do; /* See if category is within system_high */
   working_class.category = substr ("000000000000000001"b, 19 - category_number); /* get one bit */
   call convert_authorization_$to_string (working_class_bits, working_string, code);
   call convert_authorization_$to_string_short (working_class_bits, chars, code);
   chars = before (chars, " ") || "_autn";

   dirname = substr(dirname, 1, dirname_length+1) || chars;
   call try_reference_$file (dirname, "seg", null, 0, "r", "0"b, "", condition, code);
   if code ^= error_table_$incorrect_access
   then
      if code ^= 0        /* incorrect_access is the only expected status code */
      then do;
         call ioa_$rs ("Bad status code """^a""" returned on initiate of
^a segment instead of none or """^a""".", return_string, return_length,
               convert_status_code_ (code, ""), working_string, convert_status_code_ (error_table_$incorrect_access, ""));
         goto error_return;
         end;
      else                     /* no status code means this category is in our set */
         if condition ^= "" then do;  /* but there must be no condition code */
            call ioa_$rs ("Condition ^a segment signalled on read of ^a segment
instead of no condition.", return_string, return_length, condition, bit_to_integer_ (working_class.category));
            goto error_return;
            end;
         else real.category = real.category | working_class.category; /* add this bit to working_class.category set */
 end;
end;

/*  We have computed category set and authorization level.  Check with process authorization */

call hcs_$get_authorization (current_bits, max_bits);
call convert_authorization_$to_string_short (real_bits, working_string, 0);
if current.category ^= real.category | current.level ^= real.level then do;
     call convert_authorization_$to_string_short (current_bits, current_string, 0);

     /* The next two lines are necessary to combat pl1 bug 1217 */

     dcl temp1 char(100);
     temp1 = bit_to_integer_(current.category);
     call ioa_$rs ("Computed authorization does not equal process authorization!
Computed authorization: level ^d categories ^a (^a)
Process authorization: level ^d categories ^a (^a)",
                    return_string, return_length,
                    real.level, bit_to_integer_(real.category), working_string,
                    current.level, temp1, current_string);
     sw = "0"b;
     goto error_return;
     end;

if command then call ioa_ ("Process authorization is ^a", working_string);
```

```
                else message = "";
return;

error_return:

        if command then do;
                call ioa_$nnl (substr (return_string, 1, return_length));
                if sw then call ioa_ ("Error occurred when accessing ^a>sea.", dirname);
                end;
           else message = substr (return_string, 1, return_length);
end;
```

```
                    process_1_proc.pl1                      07/29/75  1342.9 edt Tue                    page 1

process_1_procl  proc (abandon_test_seg_acl);

   dcl    abandon_test_seg_acl  label;
   dcl    before  builtin;
   dcl    c_chan_id   fixed bin(71);
   dcl    channel_id  fixed bin(71);
   dcl    cleanup  condition;
   dcl    com_err_  entry options(variable);
   dcl    end_all    label internal static;
   dcl    entry_name_mailbox  char(32) initial ("test_seg_acl_mailbox");
   dcl    entry_name_workspace_dir  char(32)  initial (
                                        "test_seg_acl_workspace_dir");

   dcl    first_char  char(1);
   dcl    get_group_id_  entry returns (char(32) aligned);
   dcl    get_pdir_   entry returns (char(168) aligned);
   dcl    get_process_id_  entry  returns (bit(36));
   dcl    group_id_process_1  char(32) ;
   dcl    hcs_$add_acl_entries  entry (char(*), char(*), ptr, fixed bin,
                                     fixed bin(35));
   dcl    hcs_$append_branchx  entry (char(*),
                                    char(*),
                                    fixed bin (5),
                                    (3) fixed bin (6),
                                    char(*),
                                    fixed bin (1),
                                    fixed bin (1),
                                    fixed bin (24),
                                    fixed bin (35));
   dcl    hcs_$delentry_file  entry (char(*), char(*), fixed bin (35));
   dcl    hcs_$del_dir_tree  entry ( char(*), char(*),fixed bin (35));
   dcl    hcs_$make_seg  entry (char(*), char(*), char(*),
                             fixed bin (5),
                             ptr,
                             fixed bin (35));
   dcl    hcs_$wakeup  entry (bit(36), fixed bin(71), fixed bin(71),
                            fixed bin(35));
   dcl    ioa_  entry options(variable);
   dcl    ios_$read_ptr  entry (ptr, fixed bin, fixed bin);
   dcl    ios_$resetread  entry (char(*), bit(72) aligned);
   dcl    ipc_$create_ev_chn  entry (fixed bin(71), fixed bin(35));
   dcl    ipc_$decl_ev_call_chn  entry (fixed bin(71), entry, ptr,
                            fixed bin , fixed bin(35));

   dcl    01  wait_list_1 ,
               02  nchan  fixed bin  initial (1),
               02  channel_id (1)  fixed bin (71);
   dcl    01  mailbox_description based (ptr_mailbox),
              02  lockword  bit(36) aligned init("0"b),
              02  path_name_workspace_dir  char(168) aligned,
              02  channel_1_info,
                 03  w_chan_1_id  fixed bin(71),
                 03  c_chan_1_id  fixed bin(71),
                 03  process_1_id  bit(36),
              02  channel_2_info,
                 03  w_chan_2_id  fixed bin(71) init(0),
                 03  c_chan_2_id  fixed bin(71) init(0),
                 03  process_2_id  bit(36) init("0"b),
               02  group_id_process_2  char(32) init(""),
               02 proc_2_error_info,
                03 code  fixed bin(35),
                03 condition_found char(32),
                03 word_read bit(36),
                03 result_of_execution bit(36),
                03 ptr_try_me  ptr;
dcl   null  builtin;
dcl   num_chars  fixed bin;
dcl   path_name_mailbox  char(168) aligned;
dcl   path_name_pdir_process_1  char(168) aligned;
dcl   path_name_udd_proj_name_1  char(48);
dcl   path_name_workspace_dir  char(168) aligned;
dcl   process_1_comm_with_2  char(3) initial ("no");
dcl   process_1_proc_running  char(3) internal static initial ("no");
dcl   ptr_acl_entry_to_add  ptr;
dcl   ptr_wait_list_1  ptr;
dcl   ptr_mailbox  ptr initial (null);
dcl   ptr_response  ptr;
dcl   ring_brackets_dir (3)  fixed bin (6) internal static
                              initial (7, 7,7);
dcl   01  segment_acl  aligned ,
        02  group_id  char(32) init("*.*.*"),
        02  modes  bit(36) init("101"b),
        02  zero_pad  bit(36) init("0"b),
        02  code  fixed bin(35);
dcl   set_lock_$lock  entry (bit(36) aligned, fixed bin, fixed bin(35));
dcl   set_lock_$unlock  entry (bit(36) aligned, fixed bin(35));
dcl   status  bit(72) aligned;
dcl   status_code  fixed bin (35);
dcl   substr  builtin;
dcl   test_acl_use  entry (char(*), char(*), 1,2 bit(36) aligned,
                      2 char(168) aligned, 2, 3 fixed bin(71),
                      3 fixed bin(71), 3 bit(36), 2, 3 fixed bin(71),
                      3 fixed bin(71), 3 bit (36), 2 char(32), 2 , 3 fixed bin(35),
                      3 char(32), 3 bit(36), 3 bit(36), 3 ptr,
                      1, 2 fixed bin, 2 (1) fixed bin(71),
                      fixed bin(35));
dcl   test_add_list  entry (char(*), char(*), char(*) aligned,
                         fixed bin(35));
dcl   test_append_list  entry (char(*), char(*), char(*) aligned,
                         fixed bin(35));
dcl   test_delete_list  entry (char(*), char(*), char(*) aligned,
                         fixed bin (35));
dcl   test_replace_list  entry (char(*), char(*), char(*) aligned,
                         fixed bin (35));
dcl   user_info_  entry ( char(*), char(*), char(*));
dcl   user_1_acct  char(32);
dcl   user_1_name  char(22);
dcl   user_1_proj_id  char(9);
dcl   user_1_response char(132) aligned;



/* Do not let user_1 start process_1_proc anew without releasing prior interrupted
   run of ts_acl.
*/
If  (process_1_proc_running = "yes")
   then  do;
          call   com_err_ (0, "ts_acl",
                                 "   Release info of prior interrupted ^a",
                                 "run, before starting new run.");
            return;
         end;
   else   process_1_proc_running = "yes";



/* Construct the basic names which will be used.
*/
end_all = abandon_test_seg_acl;
path_name_pdir_process_1 = get_pdir_();
path_name_workspace_dir = before (path_name_pdir_process_1, " ")
                    || ">" || entry_name_workspace_dir;
call   user_info_ (user_1_name, user_1_proj_id, user_1_acct);
path_name_udd_proj_name_1 = ">udd>" || before (user_1_proj_id, " ")
                     || ">" || user_1_name;
path_name_mailbox = before (path_name_udd_proj_name_1, " ")
                 || ">" || entry_name_mailbox;



/* If we QUIT and release(cleanup) from here on, there may be
     workspace_dir, mailbox, channels to get rid of.
*/
on   cleanup  call  process_1_cleanup;




/* Create workspace_dir in pdir of process_1
     Giving ourselves  sma access.
*/
group_id_process_1 = get_group_id_();
call   hcs_$del_dir_tree  ((path_name_pdir_process_1),
                          entry_name_workspace_dir,
                          status_code);
call   hcs_$delentry_file ((path_name_pdir_process_1),
                          entry_name_workspace_dir,
                          status_code);
call   hcs_$append_branchx ((path_name_pdir_process_1),
                           entry_name_workspace_dir,
                            01011b,
                           ring_brackets_dir,
                           group_id_process_1,
                           1b,
                           0b,
                           0b,
                           status_code);
 if  (status_code ^= 0b)
   then  do;
            call   com_err_ (status_code,
                             "ts_acl",
                       "^/^-Could not create dir ""^a"" in process dir.",
                       entry_name_workspace_dir);

            /* From now on we must get rid of temp segs etc.
                 when exiting process_1_proc normally.
```

```
          */
          call   process_1_clearup;
          return;
        end;


/*     ************************************************
      The lacl of workspace dir should be null,
          If not, this program will have unresolvable error

       ************************************************

*/



/* Create mailbox seg in homedir of process_1.
    Giving ourselves rw_access.
*/
call   hcs_$delentry_file  (path_name_udd_proj_name_1,
                            entry_name_mailbox,
                            status_coce);
call   hcs_$make_seg  (path_name_udd_proj_name_1,
                       entry_name_mailbox,
                       "",
                       01010b,
                       ptr_mailbox,
                       status_code);
If  (status_coce ^= 0b)
  then  do;
           call   com_err_  (status_coce,
                              "ts_acl",
                   ""/^-Could nct create seg """^a""", in """^a""".",
                    entry_name_mailbox, path_name_udd_proj_name_1);
           call   process_1_cleanup;
           return;
        end;



/* Don't forget that once we created mailbox seg, it was initialized.
   Lock mailbox and fill with process_1 info.
*/
call   set_lock_$lock (ptr_mailbox -> mailbox_description.lockwcrd,
                       0b,
                       status_code);
If  (status_code ^= 0b)
 then do;
        call   com_err_  (status_code,
                              "ts_acl",
                ""/^-Could not set lock on seg """^a""".",
                path_name_mailbox);
        call  process_1_cleanup;
        return;
      end;
```

```
/* Fill mailbox with some immediate info.
*/
mailbox_description.path_name_workspace_dir = path_name_workspace_dir;
mailbox_description.process_1_id = get_process_id_ ();



/* Give everyone access to mailbox.
*/
ptr_acl_entry_to_add = addr (segment_acl);
call   hcs_$add_acl_entries (path_name_udd_proj_name_1, entry_name_mailbox,
                             ptr_acl_entry_to_add, 1, status_code);
if  (status_code ^= 0b)
   then do;
           call   com_err_ (status_code,
                            "ts_acl",
                  "^/^-Could not add ""rw  *.*.*"" to acl of ""^a"".",
                  path_name_mailbox);
           call   process_1_cleanup;
           return;
        end;



/* Create call channel 1 , we will use to wake process_1 when
     process_2 is cleaning up!
   Put  channel id into mailbox.
*/
call   ipc_$create_ev_chn (c_chan_id, status_code);
if (status_code ^= 0b)
    then do;
           call    com_err_ (0, "ts_acl",
                "   Could not create event channel , ^/^-^a (code = ^1d).",
                 "later to be converted to call channel", status_code);
           call   process_1_cleanup;
           return;
         end;
call   ipc_$decl_ev_call_chn (c_chan_id, response_call_wakeup,
                              null, 1b, status_code);
/*  Recall, response_call_wakeup does a nonlocal goto to end test_seg_acl
*/
if (status_code ^= 0b)
  then do;
         call   com_err_ (0, "ts_acl",
                "   Could not convert event channel ^a (code = ^1d).",
                "to call type channel", status_code);
         call   process_1_cleanup;
         return;
       end;


ptr_mailbox -> mailbox_description.channel_1_info.c_chan_1_id
                     = c_char_id;



/* Create wait channel 1 , we will use to wake process_1 when
     process_2 has
```

```
                1. completed an access attempt on designated seg
                   in workspace_dir.
    Put channel id into mailbox.
*/
call   ipc_$create_ev_chn (channel_id, status_code);
if  (status_code ^= 0b)
    then do;
           call   com_err_ (0, "ts_acl",
                        "   Could not create wait channel (code = ^1d).",
                       status_code);
           call   process_1_cleanup;
           return;
          end;
ptr_mailbox -> mailbox_description.channel_1_info.w_chan_1_id
                   = channel_id;



/* wait_list will later be used when process_1 goes blocked.
*/
ptr_wait_list_1 = addr (wait_list_1);
wait_list_1.channel_id(1) = channel_id;







/* OK, free mailbox and direct user_1 to create process_2
        on a ditto terminal.
*/
call   set_lock_$unlock (ptr_mailbox -> mailbox_description.lockword,
                         status_code);
if (status_code ^= 0b)
     then do;
            call   com_err_ (status_code, "ts_acl",
                          "^/^-Could not unlock seg ""^a"".",
                        path_name_mailbox);
            call   process_1_cleanup;
            return;
          end;
call   ioa_ ("*** Login at second terminal, ^a ^/^a ^a ^a "" ",
             "under a different name and/or project.",
               "*** Issue the command:  ""test_seg_acl", user_1_name, user_1_proj_id);


ptr_response = addr (user_1_response);
input_check:  call   ios_$resetread ("user_input", status);
              call   ios_$read_ptr (ptr_response, 132, num_chars);
              first_char = substr (user_1_response, 1, 1);
              if (first_char = "f")
                then do;
                       call   process_1_cleanup;
                       return;
                     end;
              if (first_char ^= "s")
                then do;
```

```
                    call   ioa_ ("!!! Did you fail(f), or ^a",
                                 "succeed(s) ?");
                    goto  input_check;
                  end;

/*   Evidently, user_1 created process_2 on new term
*/



/* If we execute the following, then process_2 has been created,
      has filled mailbox, and freed mailbox.
    Double check performance of process_2 filling mailbox.
*/
if (ptr_mailbox -> mailbox_descriptior.process_2_id = "0"b)
   then do;
          call    com_err_ (0, "ts_acl",
                  "   Seg ""^a"", not filled by process ^a.^/^a",
                  path_name_mailbox, "on other terminal",
                  "*** QUIT and release on other terminal.");
          call    process_1_cleanup;
          return;
        end;
process_1_comm_with_2 = "yes";


/* Double check that the user logged in at other terminal
    under a different name and/or project.
*/



if (group_id_process_1 = group_id_process_2)
 then do;
         call    com_err_ (0, "ts_acl",
                 "   You did not log in at other terminal, ^/^-^a",
                 "under a different name and/or project.");
         call    process_1_cleanup;
         return;
       end;




/* Now get into tests of seg acl subr

       SAC-1!
       First the mutual consistency of append, list.
*/
call   test_append_list (user_1_name, user_1_proj_id,
                         path_name_workspace_dir, status_code);
if (status_code ^= 0b)
    then do;
           call   process_1_cleanup;
           return;
         end;


```

```

/*  We can now use hcs_$append_branch

        SAC-2 to SAC-7:
      Proceed to test consistency of hcs_$add_acl_entries, hcs_$list_acl.
 */
 call   test_add_list (user_1_name, user_1_proj_id,
                    path_name_workspace_dir, status_code);
 if (status_code ^= 0b)
     then do;
           call   process_1_cleanup;
           return;
         end;



/*  We now can use hcs_$add_acl_entries

        SAC-8 to SAC-9:
      Now check consistency of delete, list.
 */
 call   test_delete_list (user_1_name, user_1_proj_id,
                    path_name_workspace_dir, status_code);
 if (status_code ^= 0b)
     then do;
           call   process_1_cleanup;
           return;
         end;



/*  We now can use hcs_$delete_acl_entries

        SAC-10 to SAC-12:
      Now check consistency of replace, list.
 */
 call   test_replace_list (user_1_name, user_1_proj_id,
                    path_name_workspace_dir, status_code);
 if (status_code ^= 0b)
    then do;
          call   process_1_cleanup;
          return;
        end;



/*  SAC-13 to SAC-25:

   Now we must construct acls for a seg. We know that list will
    show us what we want.  We must use process_2 to check that the
    acl does the job we think it should.  -ie Do we go to the correct
    entry?,   Do we have exactly the correct access we think?
 */


call   test_acl_use (user_1_name, user_1_proj_id,
                    mailbox_description, wait_list_1,
                    status_code);
```

```
if (status_code ^= 0)
    then do;
          call   process_1_cleanup;
          return;
         end;


/* All seg acl subr seem to work correctly!
*/
call   process_1_cleanup;
return;




/*  SUBROUTINES, PROCESS-1
*/

response_call_wakeup: proc;



   /*  This is invoked when process_1 is woken by process_2
          over call channel 1.

        The reason for this wakeup is that process_2_proc
          is being cleaned up !
   */

   call   com_err_ (0, "ts_acl", "   Abnormal termination caused by process ^a.",
                  "on other terminal");

   /* Do a nonlocal goto to the end of "test_seg_acl".
      This is to get us out of loc_$block, where this proc
       is called from.

      Also, it will cause activation of unwinder proc, which will
        activate cleanup in process_1_proc.
   */
   goto  end_all ;

end;


process_1_cleanup: proc;
   dcl   call_message fixed bin(71) initial (0);
   dcl   error_table_$invalid_lock_reset  fixed bin(35) external;
   dcl   error_table_$locked_by_this_process  fixed bin (35) external;
   dcl   error_table_$lock_wait_time_exceeded  fixed bin(35) external;
   dcl   five_minutes  fixed bin initial (300);
   dcl   status_code  fixed bin(35);

   if (ptr_mailbox = null)
     then do;
           /* At most we have created workspace dir.
           */
           call   hcs_$delentry_file ((path_name_pdir_process_1),
                                     entry_name_workspace_dir,
                                        status_code);
           call    hcs_$delentry_file (path_name_udd_proj_name_1,
                                        entry_name_mailbox, status_code);
           process_1_proc_running = "no";
           return;
         end;


/* We have created temporary workspace dir and mailbox
     Maybe 1. mailbox is locked, 2. 2 channels exist
*/
call    set_lock_$lock (ptr_mailbox -> mailbox_description.lockword,
                         five_minutes, status_code);
if (status_code = error_table_$locked_by_this_process)
   then do;
           /* Since process_1 locks mailbox only one time, and
              that is before process_1_comm_with_2 = "yes"
                 then we are in the midst of creating channels_1
           */
           call    std_clean;
           return;
         end;
if (status_code = error_table_$invalid_lock_reset)
   then do;
             /*  Process_2 has disappeared!
             */
             call    std_clean;
             return;
           end;
if (status_code = error_table_$lock_wait_time_exceeded)
   then do;
             /* Process_2 is holding onto mailbox for some strange
                 reason
                process_1_comm_with_2 = "no" at this point

                Likely that process_2 has been QUIT , no restart
             */
             call    std_clean;
             return;
           end;

/* status_code = 0b
   Therefore, mailbox is locked and we can delete w/o injury
     to process_2.
*/

if (process_1_comm_with_2 = "no")
   then do;
             call    std_clean;

             return;
           end;
    else do;
             call   hcs_$wakeup(ptr_mailbox ->
                                      mailbox_description.process_2_id,
                                ptr_mailbox ->
                                      mailbox_description.c_chan_2_id,
                                call_message, status_code);
                call   std_clean;
                return;
              end;

     std_clean:  proc;

               dcl   ipc_$delete_ev_chn  entry (fixed bin(71), fixed bin(35));
               dcl   ipc_$drain_chn  entry (fixed bin (71), fixed bin(35));
               dcl   status_code  fixed bin (35);
               call   hcs_$del_dir_tree ((path_name_pdir_process_1),
                                    entry_name_workspace_dir,
                                    status_code);
               call   hcs_$delentry_file ((path_name_pdir_process_1),
                                     entry_name_workspace_dir,
                                     status_code);
               call   hcs_$delentry_file (path_name_udd_proj_name_1,
                                     entry_name_mailbox, status_code);
               call   ipc_$drain_chn (wait_list_1.channel_id(1),
                                 status_code);
               call   ipc_$delete_ev_chn (wait_list_1.channel_id(1),
                                     status_code);
               call   ipc_$drain_chn (c_chan_id, status_code);
               call   ipc_$delete_ev_chn (c_chan_id,
                                     status_code);
               process_1_proc_running = "no";
               return;
     end;
   end;



end;
```

```
process_2_proc:  proc (abandon_test_seg_acl, user_1_name, user_1_proj_id);

   dcl   abandon_test_seg_acl  label;
   dcl   addr  builtin;
   dcl   before  builtin;
   dcl   c_chan_id  fixed bin(71);
   dcl   channel_id  fixed bin(71);
   dcl   cleanup  condition;
   dcl   com_err_  entry options(variable);
   dcl   condition_found  char(32) initial(" ");
   dcl   c_chan_1_id  fixed bin(71);
   dcl   end_all   label internal static;
   dcl   entry_name_mailbox  char(32) initial ("test_seg_acl_mailbox");
   dcl   entry_name_workspace_dir  char(32)  initial (
                                        "test_seg_acl_workspace_dir");
   dcl   five_minutes  fixed bin initial (300);
   dcl   get_group_id_  entry returns(char(32) aligned);
   dcl   get_process_id_  entry returns (bit(36));
   dcl   hcs_$initiate  entry (char(*), char(*), char(*), fixed bin(1),
                           fixed bin(2), ptr, fixed bin(35));
   dcl   hcs_$wakeup  entry (bit(36), fixed bin(71), fixed bin(71),
                         fixed bin(35));
   dcl   ioa_  entry options(variable);
   dcl   ipc_$block  entry (ptr, ptr, fixed bin(35));
   dcl   ipc_$create_ev_chn  entry (fixed bin(71), fixed bin(35));
   dcl   ipc_$decl_ev_call_chn  entry (fixed bin(71), entry,
                                  ptr, fixed bin, fixed bin(35));
   dcl   ipc_$delete_ev_chn  entry (fixed bin(71), fixed bin(35));
   dcl   01 wait_list_2 ,
            02  nchan  fixed bin initial (1),
            02  channel_id (1)  fixed bin(71);
   dcl   01  mailbox_description  based(ptr_mailbox),
           02  lockword  bit(36) aligned,
           02  path_name_workspace_dir  char(168) aligned,
           02  channel_1_info,
             03  w_chan_1_id  fixed bin(71),
             03  c_chan_1_id  fixed bin(71),
             03  process_1_id  bit(36),
           02  channel_2_info,
             03  w_chan_2_id  fixed bin(71),
             03  c_chan_2_id  fixed bin(71),
             03  process_2_id  bit(36),
           02  group_id_process_2  char(32),
           02  proc_2_error_info,
             03 code fixed bin(35),
             03 condition_found  char(32),
             03 word_read bit(36),
             03 result_of_execution bit(36),
             03 ptr_try_me  ptr;
   dcl   null  builtin;
   dcl   mailbox_locked_by_2  char(3)  initial ("no");
   dcl   path_name_mailbox  char(168) aligned;
   dcl   path_name_udd_proj_name_1  char(48);
   dcl   path_name_workspace_dir   char(168) aligned;
   dcl   process_1_id  bit(36);
   dcl   process_2_comm_with_1  char(3)  initial ("no");
   dcl   process_2_proc_running  char(3) internal static initial("no");
   dcl   ptr  builtin;
dcl   ptr_try_me  ptr  initial(null);
dcl   ptr_try_me_wd_1  ptr;
dcl   ptr_try_me_wd_2  ptr;
dcl   ptr_wait_list_2  ptr;
dcl   ptr_mailbox  ptr  initial (null);
dcl   ptr_wakeup_info  ptr;
dcl   result_of_execution  bit(36) aligned  initial("0"b);
dcl   set_lock_$lock  entry (bit(36) aligned, fixed bin, fixed bin(35));
dcl   set_lock_$unlock  entry (bit(36) aligned, fixed bin(35));
dcl   status_code  fixed bin(35);
dcl   try_reference_$file  entry (char(*), char(*), ptr,
                           fixed bin, char(1), bit(36) aligned,
                           char(*), char(32), fixed bin(35));
dcl   try_reference_$seg  entry (ptr, char(1), bit(36) aligned,
                              char(*), char(32), fixed bin(35));
dcl   user_1_name  char(*)  ;
dcl   user_1_proj_id  char(*)  ;
dcl   wait_message  fixed bin(71)  initial (0);
dcl   01  wakeup_info,
        02  channel_id  fixed bin(71),
        02  message  fixed bin(71) init(0),
        02  sender  bit(36),
        02  origin,
          03  dev_signal  bit(18) unaligned,
          03  ring  bit(18)  unaligned,
        02  channel_index  fixed bin;
dcl   word_read  bit(36) aligned initial("0"b);
dcl   word_to_write  bit(36) aligned initial( (36)"1"b);
dcl   word_0_of_try_me  bit(36)
          initial( "010101010101010101010101010101010101"b);
dcl   w_chan_1_id  fixed bin(71);



/* Do not let user_2 start ts_acl(x, y) without releasing prior
      interrupted run.
*/
if (process_2_proc_running = "yes")
   then do;
           call   com_err_ (0, "ts_acl",
                      "   Release info of ^a ^/^a",
                  "prior interrupted run, before starting new run.",
                  "*** Return to other terminal, type an f.");
           return;
        end;
   else
        process_2_proc_running = "yes";




/* Construct the basic names used.
*/
end_all = abandon_test_seq_acl;

path_name_udd_proj_name_1 = ">udd>" || before (user_1_proj_id, " ")
                           || ">" || before (user_1_name, " ");
path_name_mailbox = before (path_name_udd_proj_name_1, " ") ||
                ">" || entry_name_mailbox;



/* If we QUIT and rl(cleanup) from here on, then there may be ipc
      channels to get rid of.
*/
on  cleanup   call   process_2_cleanup;



/* Make mailbox known to process_2.
*/
call   hcs_$initiate (path_name_udd_proj_name_1, entry_name_mailbox,
                     "",0b, 1b, ptr_mailbox,status_code);
if (status_code ^= 0b)
    then do;
          call   com_err_ (status_code, "ts_acl",
                    "^/^-Could not initiate the seg ""^a"".^/^a",
                    path_name_mailbox,
                    "*** Return to other terminal, type an f.");
          call   process_2_cleanup;

          return;
         end;


/* Lock mailbox so that process_1 cannot destroy.
     Then proceed to fill mailbox with process_2 info.
*/
call   set_lock_$lock (ptr_mailbox -> mailbox_description.lockword,
                      five_minutes, status_code);
if (status_code ^= 0b)
  then do;
         call   com_err_ (status_code, "ts_acl",
                  "^/^-Could not set lock on seg ""^a"".^/^a",
                  path_name_mailbox,
                  "*** Return to first terminal, type an f.");
         call   process_2_cleanup;

         return;
       end;

mailbox_locked_by_2 = "yes";





/*  Process_1 now cannot do any clean up of mailbox
    Check if process_1 filled mailbox as required.
*/

if (ptr_mailbox -> mailbox_description.process_1_id = "0"b)
   then do;
          /* Process_1 did not fill mailbox for some strange reason
          */
          call    com_err_ (0, "ts_acl",
                  "   Seg """^a""", not filled by process ^a.^/^a",
                  path_name_mailbox, "on other terminal",
                  "*** Return to other terminal, type an f.");
          call    process_2_cleanup;
          return;
        end;


/*  Grab process_1 info that is in mailbox
*/
path_name_workspace_dir = ptr_mailbox -> mailbox_cescription.
                                         path_name_workspace_dir;
w_chan_1_id = ptr_mailbox -> mailbox_description.w_chan_1_id;
c_chan_1_id = ptr_mailbox -> mailbox_cescription.c_chan_1_id;
process_1_id = ptr_mailbox -> mailbox_cescription.process_1_id;
process_2_comm_with_1 = "yes";


/* Fill mailbox with some immediate info.
*/
mailbox_description.process_2_ic = get_process_id_ ();
mailbox_description.group_id_process_2 = get_group_id_ ();


/* Create call chan 2, we will use to wake process_2 if process_1
     cleaning up!
   Put channel id into mailbox.
*/
call    ipc_$create_ev_chn (c_chan_id, status_coce);
if (status_code ^= 0b)
  then do;
         call    com_err_ (0, "ts_acl",
                 "   Could not create event charnel, ^/^-^a (coce = ^id).",
                 "later to be converteo to call channel", status_code);


         /* Cleanup will abort process_1 . */
         call    process_2_cleanup;
         return;
       end;
call    ipc_$decl_ev_call_chn (c_chan_ic, response_call_wakeup,
                               null, 1b, status_code);
if (status_code ^= 0b)
  then do;
         call    com_err_ (0, "ts_acl",
                 "   Could not convert event charnel ^a (code = ^id).",
                 "to call type channel", status_code);


         /* Cleanup will abort process_1 . */
         call    process_2_cleanup;
         return;
       end;

ptr_mailbox -> mailbox_description.c_chan_2_id = c_chan_id;
```

```

/* Create wait chan 2 , we will use to wake process_2 when process_1
      has created seg in workspace_dir and wants process_2 to try
      certain type access.
   Put channel id into mailbox.
*/
call    ipc_$create_ev_chn (channel_id, status_code);
if (status_code ^= 0b)
   then do;
           call    com_err_ (0, "ts_acl",
                      "   Could not create wait charnel (code = ^id).",
                      status_code);

           /* Cleanup will abort process_1 . */
           call    process_2_cleanup;
           return;
        end;
ptr_mailbox -> mailbox_description.w_chan_2_id = channel_id;



/* wait_list will be used when process_2 goes blocked.
*/
ptr_wait_list_2 = addr (wait_list_2);
wait_list_2.channel_id (1) = channel_ic;






/* OK, free mailbox, thus process_1 car cleanup if it wants.
*/
call    set_lock_$unlock (ptr_mailbox -> mailbox_description.lockword,
                          status_code);
if (status_code ^= 0b)
   then do;
           call    com_err_ (status_code, "ts_acl",
                      "^/^-Could nct unlock seg ""^a"".", path_name_mailbox);

          /*Cleanup will abort process_1_proc */
           call    process_2_cleanup;
           return;
        end;

mailbox_locked_by_2 = "no";




/* Let process_1 know info on process_2 is in the mailbox
*/
call    ioa_ ("*** Return to first terminal, type an s.");



/* Wait for first job from process_1.
*/
```

```
ptr_wakeup_info = addr (wakeup_info);



next_wakeup:
   call   ipc_$block (ptr_wait_list_2, ptr_wakeup_info,
                         status_code);
if   (status_code ^= 0b)
  then do;
         call   com_err_ (0, "ts_acl",
                "   Could not go blocked (code = ^1d). ^/^-^a ^d.",
              status_code, "Last wakeup message = ", wakeup_info.message);
         call   process_2_cleanup;
         return;
       end;

/*
   wakeup_info.message should = 1 -> 6, where:
      1 = access to try_me should be null
      2 = r
      3 = re
      4 = rw
      5 = rew
      6 = null

   IT IS IMPORTANT to note that these wakeups appear to process_2 in time
      in the order  1,2,3,4,5,6.
*/


goto   try_access(wakeup_info.message);


/* Case 1, process_2 should have null access to try_me.
     Process_2 does not yet know of try_me.
*/
/*  SAC-13:
*/

try_access(1):  call    try_reference_$file ((path_name_workspace_dir),
                      "try_me", ptr_try_me, 0, "r", word_read, "", condition_found,
                        status_code);
                if ( (status_code = 0) | (word_read ^= "0"b) | (ptr_try_me ^= null) )
                  then do;
                      wait_message = 1000;
                      call    error_table_fill;
                      call    hcs_$wakeup (process_1_id, w_chan_1_id,
                                  wait_message, status_code);
                      call    process_2_cleanup;
                      return;
                     end;
                 call    restore;
                 wait_message = wakeup_info.message;
                 call    hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                                  status_code);
                 goto  next_wakeup;
```

```
/*  SAC-14.  SAC-18 to SAC-24:

   Case 2, process_2 should have r_access only to try_me.
    try_me may not be known to process_2.
*/
try_access(2):  call    try_reference_$file ((path_name_workspace_dir),
                         "try_me", ptr_try_me, 0, "r", word_read, "",
                         condition_found, status_code);
               if ( (status_code ^= 0) | (condition_found ^= " ") | (word_read ^= word_0_of_try_me)
                    | (ptr_try_me = null) )
                  then do;
                     wait_message = 2000;
                     call    error_table_fill;
                     call    hcs_$wakeup (process_1_id, w_chan_1_id,
                                         wait_message, status_code);
                     call    process_2_cleanup;
                     return;
                    end;
               /* Does process_2 only have r_access? */
               ptr_try_me_wd_1 = ptr (ptr_try_me, 1);
               ptr_try_me_wd_2 = ptr (ptr_try_me, 2);
               call    try_reference_$seg (ptr_try_me_wd_2, "e",
                         result_of_execution, "no_execute_permission",
                         condition_found, status_code);
               if ( (status_code ^= 0) | (condition_found ^= " ") | (result_of_execution ^= "0"b) )
                  then do;
                      wait_message = 2100;
                      call    error_table_fill;
                      call    hcs_$wakeup (process_1_id, w_chan_1_id,
                                    wait_message, status_code);
                      call    process_2_cleanup;
                      return;
                    end;
               call    try_reference_$seg (ptr_try_me_wd_1, "w", word_to_write,
                              "no_write_permission", condition_found,
                              status_code);
                if ( (status_code ^= 0) | (condition_found ^= " ") )
                  then do;
                      wait_message = 2200;
                      call    error_table_fill;
                      call    hcs_$wakeup (process_1_id, w_chan_1_id,
                                 wait_message, status_code);
                      call    process_2_cleanup;
                      return;
                    end;
               call    try_reference_$seg (ptr_try_me_wd_1, "r", word_read,
                              "", condition_found, status_code);
               if (word_read ^= "0"b)
                 then do;
                       wait_message = 2300;
                       call    error_table_fill;
                       call    hcs_$wakeup (process_1_id, w_chan_1_id,
                                  wait_message, status_code);
                       call    process_2_cleanup;
                       return;
                     end;
               /* OK, process_2 had only r_access to try_me. */
               call    restore;
```

```
                wait_message = wakeup_info.message;
                call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                                status_code);
                goto  next_wakeup;



/*  SAC-15:

    Case 3, process_2 should have re_access only tc try_me.
*/
try_access(3):
     call   try_reference_$seg (ptr_try_me, "r", word_read, "", condition_found,
                                status_code);
     if ( (status_code ^= 0) | (condition_found ^= " ") | (word_read ^= word_0_of_try_me) )
        then do:
                wait_message = 3000;
                call   error_table_fill;
                call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                                status_code);
                call   process_2_cleanup;
                return;
             end;
     call   try_reference_$seg (ptr_try_me_wd_2, "e", result_of_execution,
                              "", condition_found, status_code);
     if ( (status_code ^= 0) | (condition_found ^= " ") | (result_of_execution ^= word_0_of_try_me) )
        then do;
                wait_message = 3100;
                call   error_table_fill;
                call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                                status_code);
                call   process_2_cleanup;
                return;
              end;


     call   try_reference_$seg (ptr_try_me_wd_1, "w", word_to_write,
                 "no_write_permission", condition_found, status_code);
     if ( (status_code ^= 0) | (condition_found ^= " ") )
        then do;
                wait_message = 3200;
                call   error_table_fill;
                call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                                status_code);
                call   process_2_cleanup;
                return;
             end;
     call   try_reference_$seg (ptr_try_me_wd_1, "r", word_read, "",
                           condition_found, status_code);
     if (word_read ^= "0"b)
        then do;
                wait_message = 3300;
                call   error_table_fill;
                call   hcs_$wakeup (process_1_id, w_char_1_id, wait_message,
                                status_code);
                call   process_2_cleanup;
                return;
              end;
    /*Process_2 has only re_access to try_me
    */
    call   restore;
    wait_message = wakeup_info.message;
    call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                      status_code);
    goto   next_wakeup;


/*  SAC-16:

   Case 4, process_2 should have rw_access only to try_me.
*/
try_access(4): call   try_reference_$seg (ptr_try_me, "r", word_read,
                                      "", condition_found, status_code);
                if ( (status_code ^= 0) | (condition_found ^= " ") | (word_read ^= word_0_of_try_me) )
                 then do:
                       wait_message = 4000;
                       call   error_table_fill;
                       call   hcs_$wakeup (process_1_id, w_chan_1_id,
                                    wait_message, status_code);
                       call   process_2_cleanup;
                       return;
                     end;
                call   try_reference_$seg (ptr_try_me_wd_1, "w", word_to_write,
                                      "", condition_found, status_code);
                if ( (status_code ^= 0) | (condition_found ^= " ") )
                  then do;
                           wait_message = 4100;
                           call   error_table_fill;
                          call   hcs_$wakeup (process_1_id, w_chan_1_id,
                                         wait_message, status_code);
                          call   process_2_cleanup;
                           return;
                        end;
                call   try_reference_$seg (ptr_try_me_wd_1, "r", word_read,
                                ""、 condition_found, status_code);
                if ( word_read ^= word_to_write)
                  then do;
                          wait_message = 4200;
                           call   error_table_fill;
                          call   hcs_$wakeup (process_1_id, w_chan_1_id,
                                             wait_message, status_code);
                          call   process_2_cleanup;
                          return;
                        end;


                call   try_reference_$seg (ptr_try_me_wd_2, "e",
                          result_of_execution, "no_execute_permission",
                          condition_found, status_code);
                if ( (status_code ^= 0) | (condition_found ^= " ") | (result_of_execution ^= "0"b) )
                  then do;
                        wait_message = 4300;
                         call   error_table_fill;
                         call   hcs_$wakeup (process_1_id, w_chan_1_id,
                                   wait_message, status_code);
                         call   process_2_cleanup;
```

```
                    return;
                 end;
              /* Process_2 has only rw_access. */
              call   restore;
              call   try_reference_$seg (ptr_try_me_wd_1, "w", "0"b,
                          "", condition_found, status_code);
               wait_message = wakeup_info.message;
              call  hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                          status_code);
              goto  next_wakeup;



/*  SAC-17:

    Case 5, process_2 should have rew_access to try_me.
*/
try_access(5):
    call    try_reference_$seg (ptr_try_me, "r", word_read, "",
                        condition_found, status_code);
     if ( (status_code ^= 0) | (condition_found ^= " ") | (word_read ^= word_0_of_try_me) )
        then do;
             wait_message = 5000;
             call   error_table_fill;
             call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                           status_code);
             call   process_2_cleanup;
             return;
          end;
    call    try_reference_$seg (ptr_try_me_wd_2, "e", result_of_execution,
                        "", condition_found, status_code);
    if ( (status_code ^= 0) | (condition_found ^= " ") | (result_of_execution ^= word_0_of_try_me) )
        then do;
              wait_message = 5100;
              call   error_table_fill;
              call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                          status_code);
              call   process_2_cleanup;
              return;
            end;
    call    try_reference_$seg (ptr_try_me_wd_1, "w", word_to_write,
                        "", condition_found, status_code);
    if ( (status_code ^= 0) | (condition_found ^= " ") )
        then do;
             wait_message = 5200;
             call   error_table_fill;
             call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                           status_code);
             call   process_2_cleanup;
             return;
          end;
    call    try_reference_$seg (ptr_try_me_wd_1, "r", word_read, "",
                        condition_found, status_code);
    if (word_read ^= word_to_write)
     then do;
            wait_message = 5300;
            call   error_table_fill;
            call   hcs_$wakeup ( process_1_id, w_chan_1_id, wait_message,
                        status_code);
          call   process_2_cleanup;
          return;
        end;
   /* Process_2 had exactly rew_access */
   call   restore;
   call   try_reference_$seg (ptr_try_me_wd_1, "w", "0"b, "",
                          condition_found, status_code);
   wait_message = wakeup_info.message;
   call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message, status_code);
   goto   next_wakeup;


/*  SAC-25:

   Case 6, process_2 should have no access to try_me, because its
    last acl entry has been removed.
     Such a change effectively terminates try_me from process_2.
*/
try_access(6):
      call   try_reference_$seg (ptr_try_me, "r", word_read, "seg_fault_error",
                          condition_found, status_code);
    if ( (status_code ^= 0) | (condition_found ^= " ") | (word_read ^= "0"b) )
       then do;
              wait_message = 6000;
              call   error_table_fill;
               call   hcs_$wakeup (process_1_id, w_chan_1_id,
                              wait_message, status_code);
               call   process_2_cleanup;
               return;
            end;

    call   restore;
     wait_message = wakeup_info.message;
     call   hcs_$wakeup (process_1_id, w_chan_1_id, wait_message,
                      status_code);
     goto  next_wakeup;



  /*  SUBROUTINES  PROCESS-2
  */


  error_table_fill:  proc;

        dcl   code  fixed bin (35);

        call   set_lock_$lock (mailbox_description.lockword, five_minutes, code );
        if (code ^= 0)
              then  /*  Process_1 is likely gone !!!! */
                    return;
        mailbox_description.code = status_code;
        mailbox_description.condition_found = condition_found;
        mailbox_description.word_read = word_read;
        mailbox_description.result_of_execution = result_of_execution;
        mailbox_description.ptr_try_me = ptr_try_me;
        call    set_lock_$unlock (mailbox_description.lockword, code);

end;


restore: proc;

        condition_found = " ";
        word_read = "0"b;
        result_of_execution = "0"b;

end;



response_call_wakeup:  proc;


      /* This is invoked when process_2  is woken
         by process_1 over call channel 2

         The reason for this wakeup is that process_1
          is being cleaned up !
      */



      /* Do a nonlocal goto to end of test_seg_acl.
          This will get us out of ipc_$block which calls this proc.
         Also, it will cause the activation of the unwinder proc,
           which will activate the cleanup in process_2_proc.
      */
      goto   end_all;
end;

process_2_cleanup:  proc;

      dcl   call_message   fixed bin (71) initial (0);
      dcl   hcs_$terminate_seg  entry (ptr, fixed bin(1), fixed bin(35));
      dcl   ipc_$drain_chn  entry (fixed bin(71), fixed bin(35));
      dcl   status_code  fixed bin(35);



      /* Locate where you were in the process_2_proc, then cleanup.
      */
      if (process_2_comm_with_1 = "yes")
        then
            if (mailbox_locked_by_2 = "yes")
              then do;
                    call   hcs_$wakeup (process_1_id, c_chan_1_id,
                                    call_message, status_code);
                    call   set_lock_$unlock (ptr_mailbox ->
                                      mailbox_description.lockword,
                                      status_code);
                    call  ipc_$drain_chn (wait_list_2.channel_id(1),
                                    status_code);
                    call ipc_$delete_ev_chn (wait_list_2.channel_id
                                (1), status_code);
                call  ipc_$drain_chn (c_chan_id, status_code);
                call ipc_$delete_ev_chn (c_chan_id, status_code);
                call   hcs_$terminate_seg (ptr_mailbox, 0,
                                status_code);
                process_2_proc_running = "no";
                return;
              end;
          else do;
                call   hcs_$wakeup (process_1_id, c_chan_1_id,
                             call_message, status_code);
                call ipc_$drain_chn (wait_list_2.channel_id(1),
                           status_code);
                call ipc_$delete_ev_chn (wait_list_2.channel_id
                                (1), status_code);
                 call ipc_$drain_chn (c_chan_id, status_code);
                call ipc_$delete_ev_chn (c_chan_id, status_code);
                call   hcs_$terminate_seg (ptr_mailbox, 0,
                             status_code);
                call   hcs_$terminate_seg (ptr_try_me, 0, status_code);
                process_2_proc_running = "no";
                return;
              end;

     else
        if (mailbox_locked_by_2 = "yes")
           then do;
                  call   set_lock_$unlock (ptr_mailbox ->
                               mailbox_descriptior.lockword,
                                 status_code);
                 call  hcs_$terminate_seg (ptr_mailbox, 0,
                                status_code);
                  process_2_proc_running = "no";
                  return;
               end;
           else do;
                  call  hcs_$terminate_seg (ptr_mailbox, 0,
                                 status_code);
                  process_2_proc_running = "no";
                  return;
               end;


   end;
end;
```

```
response_to_start_up:  proc;

    dcl   before  builtin;
    dcl   cleanup  condition;
    dcl   code  fixed bin(35);
    dcl   com_err_  entry options(variable);
    dcl   convert_authorization_$from_string  entry (bit(72) aligned,
                                        char(*), fixed bin(35));
    dcl   convert_authorization_$to_string_short  entry (bit(72) aligned,
                                        char(*), fixed bin(35));
    dcl   entry_name  char(32)  init("multi_process_info");
    dcl   error_table_$wakeup_denied   fixed bin(35) external;
    dcl   hcs_$get_authorization  entry (bit(72) aligned, bit(72) aligned);
    dcl   hcs_$initiate  entry (char(*), char(*), char(*), fixed bin(1),
                         fixed bin(2), ptr, fixed bin(35));
    dcl   hcs_$wakeup  entry (bit(36), fixed bin(71), fixed bin(71),
                        fixed bin(35));
    dcl   hdir  char(68);
    dcl   01 info  aligned based(ptr_info),
             02 process_2_id bit(36),
             02 channel_id  fixed bin(71),
             02 authorization_1 bit(72) aligned,
             02 authorization_2 bit(72) aligned,
             02 authorization_3 bit(72) aligned,
             02 authorization_4 bit(72) aligned,
             02 authorization_5 bit(72) aligned,
             02 authorization_6 bit(72) aligned;
    dcl   message  fixed bin(71);
    dcl   new_proc_  entry (bit(72) aligned, fixed bin(35));
    dcl   next_authorization_bit  bit(72) aligned;




    /* ********************

          On all the authorizations_char: it is possible that 150 is too short.
          If program gets into weird errors, think of this.

    */
    dcl   next_authorization_char  char(150);
    dcl   null    builtin;
    dcl   path_name_info_seg  char(168);
    dcl   present_authorization_bit  bit(72) aligned ;
    dcl   present_authorization_char  char(150);
    dcl   present_max_authorization_bit  bit(72) aligned;
    dcl   problem_with_wakeup  fixed bin  init(0);
    dcl   ptr_info  ptr init(null);
    dcl   system_low_bit  bit(72) aligned;
    dcl   user_info_$homedir  entry (char(*));




    /* Get the process authorization.
    */
    call  hcs_$get_authorization (present_authorization_bit,
                        present_max_authorization_bit);




/* For later reference, get the bit str rep. of "system_low".
     This routine should not fail !!!!!
*/
call   convert_authorization_$from_string (system_low_bit,
                                 "system_low", code);



/* It will be useful to have present authorization in char form.
     This call should not fail.
*/
call   convert_authorization_$to_string_short (present_authorization_bit,
                                 present_authorization_char, code);
on  cleanup  call   response_clean;


/* Initiate the all important info seg.
*/
call   user_info_$homedir (hdir);


path_name_info_seg = before ( hdir, " ") || ">" || entry_name;


call   hcs_$initiate (hdir, entry_name, "", 0, 1, ptr_info, code);
if (code ^= 0)
 then do;
        problem_with_wakeup = 1;
        call   com_err_ (code, "tipc", "^/At authorization = ^a, could not initiate segment: ^/^-^a",
                         present_authorization_char, path_name_info_seg);
        call   response_clean;
        return;
      end;




/*  Check quickly to see if info seg is filled.
 */
if (process_2_id = "0"b)
   then do;
          problem_with_wakeup = 1;
          call   com_err_ (0, "tipc", "At authorization = ^a, segment ^a ^/^-^a",
                          present_authorization_char, path_name_info_seg, "is missing ipc information.");
          call   response_clean;
          return;
        end;


/* 1. Send a wakeup message to process_2 on term_2.
   2. If we are not now at system_low, then new proc to "next" authorization
        that is in our test sequence.
*/
```

```


/* Determine  correct wakeup msg and "next" authorization to new_proc_ to
*/
if (present_authorization_bit = authorization_1)
   then do; message = 1; next_authorization_bit = authorization_2; end;
else  if (present_authorization_bit = authorization_2)
        then do; message = 2; next_authorization_bit = authorization_3; end;
else  if (present_authorization_bit = authorization_3)
        then do; message = 3; next_authorization_bit = authorization_4; end;
else  if (present_authorization_bit = authorization_4)
        then do; message = 4; next_authorization_bit = authorization_5; end;
else  if (present_authorization_bit = authorization_5)
        then do; message = 5; next_authorization_bit = authorization_6; end;
else  if (present_authorization_bit = authorization_6)
        then do; message = 6; next_authorization_bit = system_low_bit; end;
else  if (present_authorization_bit = system_low_bit)
        then  do; message = 7; next_authorization_bit = system_low_bit; end;
else  /* Present auth is not one which is expected !
      */
       do:
           call   com_err_ (0, "tipc", "At an unexpected authorization !");
           next_authorization_bit = system_low_bit;
           call   new_proc_ (next_authorization_bit, code);
           if (code ^= 0)
                 then do;
                              call   com_err_ (code, "tipc", "^/At authorization = ^a, ^/^-^a ^a",
                                   present_authorization_char, "could not new_proc_ to authorization = ",
                                   "system_low");
                              call   response_clean;
                              return;
                         end;

       end;




/* For later use:
*/
call   convert_authorization_$to_string_short (next_authorization_bit,
                                       next_authorization_char, code);
if (code ^= 0)
  then do:
        call   com_err_ (code, "tipc", "At authorization = ^a, ^/^-^a",
         present_authorization_char, "could not convert next authorization to character form.");
        call   response_clean;
        return;
      end;


/* Get the message off to process_2 on terminal 2.
       The way this subroutine is called, these
       calls occur in order as IPC-2, IPC-3, IPC-1
       IPC-4, IPC-5, IPC-6.
*/
call   hcs_$wakeup ((process_2_id), channel_id, message, code);
if ( (message = 1) | (message = 2) | (message = 3) | (message = 7) )
```

```
then if (code ^= 0)
        then do;
                problem_with_wakeup = 1;
                call    com_err_ (code, "tlpc", "At authorization = ^a, ^a^/^-^a. ",
                             present_authorization_char, "failed to wakeup",
                             "process on other terminal");
                call    response_clean;
                return;
              end;
          else ;

else if ( code ^= error_table_$wakeup_denied)
          then do;
                    problem_with_wakeup = 1;
                    call    com_err_ (code, "tlpc", "At authorization = ^a, ^a^/^-^a.",
                                        present_authorization_char, "did not receive the ",
                                        "code ""wakeup_denied"" upon attempt to wake process on other terminal");
                    call    response_clean;
                    return;
              end;





/* If ok, new_proc_ out of here!!
*/
if (present_authorization_bit ^= system_low_bit)
  then do;
        call    new_proc_ (next_authorization_bit, code):
        if (code ^= 0)
         then do;
                call com_err_ (code, "tlpc", "At authorizatlor = ^a, ^/^-^a ^a",
             present_authorization_char, "could not new_proc_ to authorization = ",
                        next_authorization_char):
                call    response_clean:
                return;
              end;
        end;



/* OK, present authorization = "system_low".
        1. We have woker process_2 with last msg.
        2. That is all we have to do.
*/
call    response_clean;
return;


response_clean:  proc;

  dcl    hcs_$delentry_seg  entry (ptr, fixed bin(35));
  dcl    loa_  entry options(variable);
  dcl    status_code  fixed bin(35);

  if (present_authorization_bit = system_low_bit)
```

```
        then do;
              if (problem_with_wakeup = 1)
               then do;
                     call    ioa_ ("^/*** QUIT and release on other terminal.");
                     call    hcs_$delentry_seg (ptr_info, status_code):
                     if (status_code ^= 0)
                       then  call    ioa_ ("^/***  Segment ^a  still exists ,and should be deleted.",
                                         path_name_info_seg);
                      return;
                    end;
               else do;
                      message = 7;
                      call    hcs_$wakeup ((process_2_id), channel_id,
                                         message, status_code);
                      call    hcs_$delentry_seg (ptr_info, status_code);
                      if (status_code ^= 0)
                        then call  ioa_ ("^/*** Segment ^a still exists , and should be deleted.",
                                         path_name_info_seg);
                      return;
                    end;
              end;


     /* We were not at system_low.
     */
       call    new_proc_ (system_low_bit, status_code):
       if (status_code ^= 0)
         then do;
               call    ioa_ ("^/ ERROR: You have been left at authorization = ^a",
                            present_authorization_char);
               call    hcs_$delentry_seg (ptr_info, status_code):
               if (status_code ^= 0)
                then    call    ioa_ ("^/*** Segment ^a still exists , and should be deleted.",
                                         path_name_info_seg):
               call    ioa_ ("^/*** QUIT and release on other terminal."):
              end;
    end;

end;
```

```
/* Access Isolation Test for segments and directories.  Series SSC and DSC.

   Note: After making any changes to the source of this program,
   the line_number_inserter should be run to insert the line numbers
   in the calls to set_saved_loc. */

test_seg_autht: tsat: proc;
dcl number_ entry (fixed bin(35)) returns (char(*));
dcl get_dir_arg_ entry (fixed bin, char(*), fixed bin(35));
dcl get_group_id_ entry returns (char(32) aligned);
dcl get_pdir_ entry returns (char(168) aligned);
dcl error_table_$incorrect_access external fixed bin(35);
dcl error_table_$no_info external fixed bin(35);
dcl error_table_$ai_restricted external fixed bin(35);
dcl try_reference_$seg entry (ptr, char(1), bit(36) aligned, char(*), char(32), fixed bin(35));
dcl try_dir_reference_ entry (char(*), char(*), char(*), char(*), bit(1), fixed bin(35));
dcl try_reference_$file entry (char(*), char(*), ptr, fixed bin, char(1), bit(36) aligned, char(*),
          char(32), fixed bin(35));
dcl convert_status_code_ entry (fixed bin(35), char(8) aligned) returns (char(100) aligned);
dcl (ioa_, ioa_$nnl) entry options (variable);
dcl unique_bits_ entry returns (bit(70));
dcl dirname char(168);
dcl dirpath char(168) varying;
dcl code fixed bin(35);
dcl saved_loc fixed bin; /* line number of last test */
dcl saved_name char(10); /* number (name) of last test */
dcl 1 access_class aligned based,
          2 category bit(36) unaligned,
          2 level fixed bin(17) unaligned,
          2 pad2 bit(18) unaligned;
dcl 1 (current, max, lower, higher, class) like access_class;
dcl (current_bits based (addr(current)),
     max_bits based (addr(max)),
     lower_bits based (addr(lower)),
     higher_bits based (addr(higher)),
     system_low_bits,
     class_bits based (addr(class)))              aligned bit(72);

%include create_branch_info;
dcl 1 branch_ like create_branch_info aligned;

dcl (s1, s2) char(150);
dcl com_err_ entry options(variable);
dcl hcs_$delentry_file entry (char(*), char(*), fixed bin(35));
dcl hcs_$get_authorization entry (bit(72) aligned, bit(72) aligned);
dcl hcs_$status_ entry (char(*), char(*), fixed bin(1), ptr, ptr, fixed bin(35));
dcl hcs_$get_access_class entry (char(*), char(*), bit(72) aligned, fixed bin(35));
dcl hcs_$create_branch_ entry (char(*), char(*), ptr, fixed bin(35));
dcl convert_authorization_$from_string entry (bit(72) aligned, char(*), fixed bin(35));
dcl convert_authorization_$to_string entry (bit(72) aligned, char(*), fixed bin(35));
dcl word bit(36) based;
dcl zeros_and_ones bit(36) static init((18)"01"b);
dcl pname char(32);
dcl segptr ptr;
dcl null builtin;
dcl condition_name char(32);
dcl (bits, saved_bits) bit(36) aligned;
dcl who char(32);
dcl who_code bit(1); /* 0 if tsa, 1 if tda */
dcl i fixed bin;
dcl cleanup condition;
dcl 1 saved_status (18:20) aligned,    /* holds status of certain branches at beginning of test */
      2 (type bit(2),              /* for later check that the status has not been implicitly modified */
         name bit(18),             /* status(18) for DSC-18 has status of equal_equal (dirs_to_try(1)) */
         dim bit(36),              /* status(19) for DSC-19 has status of lower_equal>dir */
         dtu bit(36),              /* status(20) for DSC-20 has status of lower_equal>seg */
         mode bit(5),
         pad bit(13),
         records bit(18)) unaligned;

dcl 1 test_status like saved_status;

dcl pathnames (18:20) char(168) varying;
dcl enames (18:20) char(32) varying;

/* Names of directories and access modes expected for the segment access tests
   and first six directory access tests */

dcl dirs_to_try(6) char(32) varying static init ("equal_equal",
                                    "lower_equal",
                                    "higher_equal",
                                    "equal_subset",
                                    "equal_superset",
                                    "equal_isolated");
dcl upgrade_bits(6) bit(1) static init ("0"b, "0"b, "1"b, "0"b, "1"b, "1"b);
dcl expected_mode(6) char(2) init ("sm", "s", "n", "s", "n", "n");

/* Set code indicating who was called */

   who = "test_seg_auth";
   who_code = "0"b;
   goto common;

test_dir_auth: tda: entry;

   who = "test_dir_auth";
   who_code = "1"b;

common:

 call get_dir_arg_ (1, dirname, code);
if code ^= 0 then do;
   call com_err_ (code, who, dirname);
   return;
   end;
dirpath = substr (dirname, 1, 169-verify(reverse(dirname)," "));
call hcs_$get_authorization (current_bits, max_bits);

/* First check if all the directories exist, and if their access classes
   bear the proper relationship to the current authorization. */

do i = 1 to 6;
     call hcs_$get_access_class (dirname, (dirs_to_try(i)), class_bits, code);
     if code ^= 0 then do;
          call com_err_ (code, who, "^a>^a", dirname, dirs_to_try(i));
          call ioa_ ("^a: Couldn't perform any tests.", who);
```

```
            return;
            end;
        go to case(i);

        case(1): if class_bits ^= current_bits then call bad_dir(i); goto loop;
        case(2): if current.level <= class.level | current.category ^= class.category then call bad_dir(i); goto loop;
        case(3): if current.level >= class.level | current.category ^= class.category then call bad_dir(i); goto loop;
        case(4): if current.level ^= class.level | (class.category & ^current.category) ^= ""b then call bad_dir(i); goto loop;
        case(5): if current.level ^= class.level | (current.category & ^class.category) ^= ""b then call bad_dir(i); goto loop;
        case(6): if current.level ^= class.level | (current.category & ^class.category) = ""b
                    | (class.category & ^current.category) = ""b then call bad_dir(i); goto loop;
        loop: if i = 4 then lower_bits = class_bits;
              if i = 5 then higher_bits = class_bits;
end;
/*
```

```
*/
132 if who_code = "0"b then do;    /* Make segment tests */
133
134 /* SSC-1: First test initiates segment of same authorization level and category and
135           writes a unique bit string into word 1 */
136
137           call set_saved_loc ( 137, "SSC-1");
138           pname = dirs_to_try(1) || ">dir>seg";
139           saved_bits = substr (unique_bits_(), 1, 36);
140           call try_reference_$file (dirpath || ">" || dirs_to_try(1) || ">dir", "seg", segptr, 1, "w", saved_bits,
141                   "", condition_name, code);
142           if segptr = null then
143 initiate_error: call code_error ("initiate", code, pname, "no error.");
144           if condition_name ^= "" then
145 write_error:    call condition_error ("write", condition_name, pname, "none");
146
147 /* SSC-2: Read the word just written and see if it is OK */
148
149           call set_saved_loc ( 149, "SSC-2");
150           segptr = ptr(segptr, 1);
151           call try_reference_$seg (segptr, "r", bits, "", condition_name, code);
152           if condition_name ^= "" then
153 read_error:     call condition_error ("read", condition_name, pname, "none");
154           if bits ^= saved_bits then do;
155                call ioa_ ("^/Data read from segment ^a>^a^/is not the same as data written.",
156                        dirname, pname);
157                goto error_return;
158                end;
159
160 /* SSC-3: Try to execute segment of same access class and see if it works */
161
162           call set_saved_loc ( 162, "SSC-3");
163           bits = ""b;
164           call try_reference_$seg (ptr (segptr, 2), "e", bits, "", condition_name, code);
165           if condition_name ^= "" then
166 execute_error:  call condition_error ("execute", condition_name, pname, "none");
167           if bits ^= zeros_and_ones then do;
168 bad_execute:    call ioa_ ("^/Program in ^a>^a was called and^/returned but did not execute properly", dirname, pname);
169                goto error_return;
170                end;
171
172 /* SSC-4: The next few tests work with a segment of a lower level but same category */
173
174           call set_saved_loc ( 174, "SSC-4");
175           pname = dirs_to_try(2) || ">dir>seg";
176           call try_reference_$file (dirpath || ">" || dirs_to_try(2) || ">dir", "seg", segptr, 0, "r", bits, "",
177                   condition_name, code);
178           if segptr = null then goto initiate_error;
179           if condition_name ^= "" then goto read_error;
180           if bits ^= zeros_and_ones then do;
181 bad_read:       call ioa_ ("^/Read of segment ^a>^a^/was allowed but data read was bad.", dirname, pname);
182                goto error_return;
183                end;
184
185 /* SSC-5: Try to execute segment of lower level */
186
187           call set_saved_loc ( 187, "SSC-5");
188           bits = ""b;
```

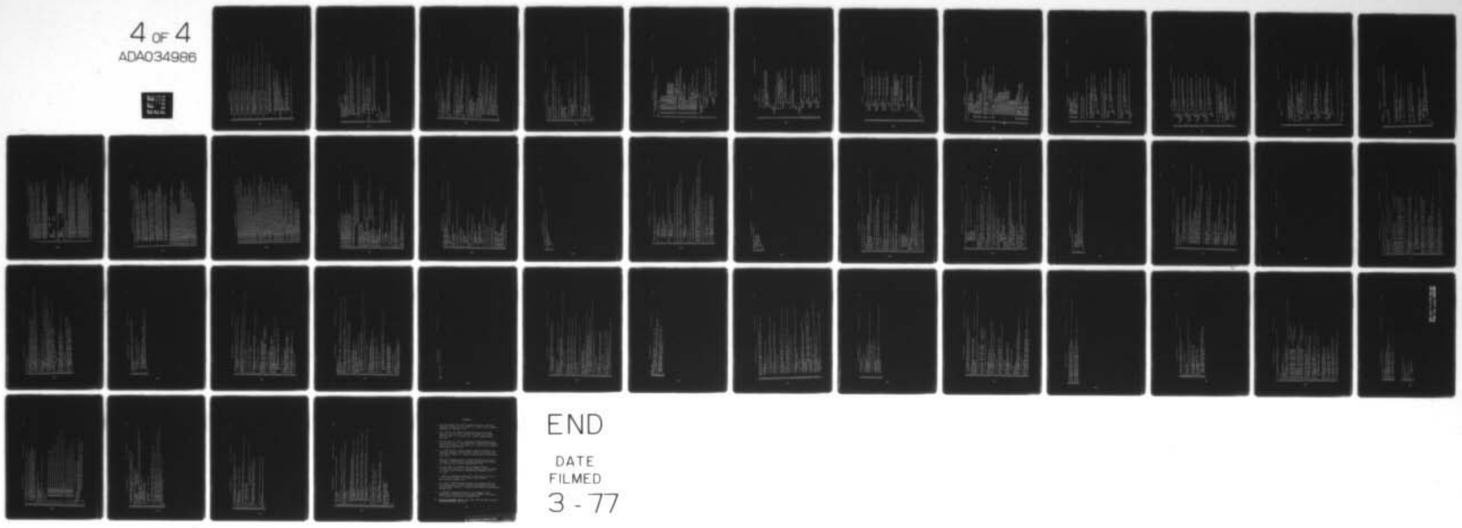
```
189          call try_reference_$seg (ptr (segptr, 2), "e", bits, "", condition_name, code);
190          if condition_name ^= "" then goto execute_error;
191          if bits ^= zeros_and_ones then goto bad_execute;
192
193 /* SSC-6: Try to write segment of lower level, which should be illegal */
194
195          call set_saved_loc ( 195, "SSC-6");
196          call try_reference_$seg (ptr (segptr, 1), "w", saved_bits, "no_write_permission", condition_name, code);
197          if condition_name ^= "" then
198 no_write_error: call condition_error ("write", condition_name, pname, "no_write_permission");
199
200          /* Even if the right condition occurred, make sure that the write didn't work */
201
202          if ptr (segptr, 1) -> word = saved_bits then do;
203 bad_write:      call ioa_ ("^/Illegal write was allowed on segment ^a>^a^/even though ""no_write_permission"" was signalled",
204                     dirname, pname);
205                goto error_return;
206                end;
207
208 /* SSC-7: Try references on seg, which has a higher level and same category */
209
210          call set_saved_loc ( 210, "SSC-7");
211          pname = dirs_to_try(3) || ">dir>seg";
212          call try_reference_$file (dirpath || ">" || dirs_to_try(3) || ">dir", "seg", segptr, 0, "r", bits, "x",
213                    condition_name, code);
214          if code ^= error_table_$incorrect_access then do;
215 moderr:         if code ^= 0 then
216                    call code_error ("initiate", code, pname,
217                              convert_status_code_ (error_table_$incorrect_access, ""));
218                call ioa_ ("Initiate of segment ^a>^a returned no status code.
219 Instead, condition ""^a"" occurred on attempted read.", dirname, pname, condition_name);
220                goto error_return;
221                end;
222          if segptr ^= null then do;
223 no_null_ptr:    call ioa_ ("^/Status code ""^a"" returned^/on initiate of ^a>^a, but
224 pointer to segment was not null.  Pointer = ^p.", convert_status_code_ (error_table_$incorrect_access, ""),
225                     dirname, pname, segptr);
226                goto error_return;
227                end;
228
229 /* SSC-8: Try to read seg which has same level but fewer categories */
230
231          call set_saved_loc ( 231, "SSC-8");
232          pname = dirs_to_try(4) || ">dir>seg";
233          bits = ""b;
234          call try_reference_$file (dirpath || ">" || dirs_to_try(4) || ">dir", "seg", segptr, 0, "r", bits,
235                    "", condition_name, code);
236          if segptr = null then goto initiate_error;
237          if condition_name ^= "" then goto read_error;
238          if bits ^= zeros_and_ones then goto bad_read;
239
240 /* SSC-9: Execute seg */
241
242          call set_saved_loc ( 242, "SSC-9");
243          bits = ""b;
244          call try_reference_$seg (ptr (segptr, 2), "e", bits, "", condition_name, code);
245          if condition_name ^= "" then goto execute_error;
246          if bits ^= zeros_and_ones then goto bad_execute;
```

```
247
248 /* SSC-10: Write seg, which should be illegal */
249
250         call set_saved_loc ( 250, "SSC-10");
251         call try_reference_$seg (ptr (segptr, 1), "w", saved_bits, "no_write_permission",
252                 condition_name, code);
253         if condition_name ^= "" then goto no_write_error;
254         if ptr (segptr, 1) -> word = saved_bits then goto bad_write;
255
256 /* SSC-11: This test is initiate of seg, which has more categories but same level */
257
258         call set_saved_loc ( 258, "SSC-11");
259         pname = dirs_to_try(5) || ">dir>seg";
260         call try_reference_$file (dirpath || ">" || dirs_to_try(5) || ">dir", "seg", segptr, 0, "r", bits,
261                 "x", condition_name, code);
262         if code ^= error_table_$incorrect_access then goto moderr;
263         if segptr ^= null then goto no_null_ptr;
264
265 /* SSC-12: Final test is isolated category sets, same level */
266
267         call set_saved_loc ( 267, "SSC-12");
268         pname = dirs_to_try(6) || ">dir>seg";
269         call try_reference_$file (dirpath || ">" || dirs_to_try(6) || ">dir", "seg", segptr, 0, "r", bits,
270                 "x", condition_name, code);
271         if code ^= error_table_$incorrect_access then goto moderr;
272         if segptr ^= null then goto no_null_ptr;
273         end;
274 /*
```

```
*/
275 if who_code = "1"b then do;   /* Make directory tests */
276
277 /* Before making any tests, save the status of three entries whose dtm and dtu should
278    not be modified by these tests */
279
280      pathnames(18) = dirpath; enames(18) = dirs_to_try(1);
281      pathnames(19) = dirpath || ">" || dirs_to_try(2); enames(19) = "dir";
282      pathnames(20) = dirpath || ">" || dirs_to_try(2); enames(20) = "seg";
283
284      do i = 18 to 20;
285           call hcs_$status_ ((pathnames(i)), (enames(i)), 0, addr(saved_status(i)), null(), code);
286           if code ^= 0 then do;
287                call com_err_ (code, who, "^a>^a", pathnames(i), enames(i));
288                call ioa_ ("^a: Can't make directory tests because status of^/^a>^a is not available.", who, pathnames(i), enames(i
));
289                return;
290                end;
291      end;
292
293 /* DSC-1 to DSC-6: Check each of the directories of different levels */
294
295      do i = 1 to 6;
296           call set_saved_loc ( 296, "DSC-" || number_ ((i)));
297           call try_dir_reference_ (dirpath || ">" || dirs_to_try(i), "dir",
298                     "seg", expected_mode(i), upgrade_bits(i), code);
299           if code ^= 0 then goto error_return:
300      end;
301
302 /* Initialize to test the upgrade primitive */
303
304      branch_.version = create_branch_version_1;
305      branch_.mode = "111"b;
306      branch_.switches = ""b;
307      branch_.rings = 7;
308      branch_.userid = get_group_id_():
309      branch_.dir_sw = "1"b;
310      branch_.mbz2 = ""b;
311      branch_.bitcnt = 0;
312      call convert_authorization_$from_string (system_low_bits, "system_low", code); /* get system_low */
313
314 /* DSC-7: Try to create upgraded directory of current authorization in lower level directory */
315
316      call set_saved_loc ( 316, "DSC-7");
317      call upgrade (dirs_to_try(2), current_bits, 1, error_table_$incorrect_access):
318
319 /* DSC-8: Try to create upgraded directory of lower category in lower level directory */
320
321      call set_saved_loc ( 321, "DSC-8");
322      call upgrade (dirs_to_try(2), lower_bits, 1, error_table_$incorrect_access);
323
324 /* DSC-9: Try to create upgraded directory of lower category set in lower category directory */
325
326      call set_saved_loc ( 326, "DSC-9");
327      call upgrade (dirs_to_try(4), lower_bits, 1, error_table_$incorrect_access);
328
329 /* DSC-10: Try to create upgraded directory of current authorization in higher level directory */
330
```

AD-A034 986   MITRE CORP BEDFORD MASS   F/G 15/3
TEST PROCEDURES FOR MULTICS SECURITY ENHANCEMENTS,(U)
DEC 76 M GASSER, S R AMES, L J CHMURA   F19628-76-C-0001
UNCLASSIFIED   MTR-3005   ESD-TR-76-164   NL

4 OF 4
ADA034986



END
DATE
FILMED
3-77

```
    call set_saved_loc ( 331, "DSC-10");
    call upgrade (dirs_to_try(3), current_bits, 1, error_table_$incorrect_access);

/* DSC-11: Try to create upgraded directory of current authorization in higher category directory */

    call set_saved_loc ( 336, "DSC-11");
    call upgrade (dirs_to_try(5), current_bits, 1, error_table_$incorrect_access);

/* DSC-12: Try to create upgraded directory of lower category in current authorization directory */

    call set_saved_loc ( 341, "DSC-12");
    call upgrade (dirs_to_try(1), lower_bits, 1, error_table_$ai_restricted);

/* DSC-13: Try to create upgraded directory of system_low in current authorization directory */

    call set_saved_loc ( 346, "DSC-13");
    call upgrade (dirs_to_try(1), system_low_bits, 1, error_table_$ai_restricted);

/* DSC-14: Try to create upgraded directory of current authorization in current authorization directory */

    on cleanup call cleanup_proc;
    call set_saved_loc ( 352, "DSC-14");
    call upgrade (dirs_to_try(1), current_bits, 1, 0); /* This should work */
    call hcs_$delentry_file (dirpath || ">" || dirs_to_try(1), "upgraded_dir", 0); /* delete it again */

/* DSC-15: Try to create upgraded directory of higher category in current authorization directory with zero quota */

    call set_saved_loc ( 358, "DSC-15");
    call upgrade (dirs_to_try(1), higher_bits, 0, error_table_$ai_restricted);

/* DSC-16: Try to create upgraded directory of higher category in current authorization directory ([pd]) with quota */

    call set_saved_loc ( 363, "DSC-16");
    call upgrade ("", higher_bits, 1, 0); /* this operation is completely legal */

    /* The last upgrade operation should have worked.  If it did, get the access_class of the new
       directory and see if it is that expected.  We don't really trust this
       value, though, so we try try_dir_reference_ to see if it is really upgraded.
    */

    call hcs_$get_access_class (get_pdir_(), "upgraded_dir", class_bits, code);
    if class_bits ^= higher_bits & code = 0 then do;
         call convert_authorization_$to_string (class_bits, s1, code);
         call convert_authorization_$to_string (higher_bits, s2, code);
         call ioa_ ("^/Directory ^a>upgraded_dir^/of access class
^a was created,^/instead of access class ^a
as specified in the call to hcs_$create_branch_.",
         get_pdir_(), s1, s2);
         goto error_return;
         end;
     if code = 0 then do;
        call try_dir_reference_ (before(get_pdir_(), " ") || ">" ||
         "upgraded_dir", "dir", "seg", "n", "1"b, code);
        if code ^= 0 then goto error_return;
        end;

/* DSC-17: Try a privileged upgrade of a segment.  This should be illegal */
```

```
389      branch_.priv_upgrade_sw = "1"b;
390      branch_.dir_sw = "0"b;
391      branch_.quota = 0;
392      branch_.access_class = higher_bits;
393      call set_saved_loc ( 393, "DSC-17");
394      call hcs_$create_branch_ (dirpath || ">" || dirs_to_try(i), "upgraded_seg", addr(branch_), code);
395      if code ^= error_table_$ai_restricted then do;
396         if code ^= 0
397         then call ioa_ ("^/Status code ""^a""^/was returned on attempt to", convert_status_code_ (code, ""));
398         else call ioa_ ("^/No status code was returned on attempt to");
399         call ioa_ ("create the upgraded segment ^a>^a>upgraded_seg", dirpath, dirs_to_try(i));
400         call ioa_ ("from ring 4, using hcs_$create_branch_.");
401         goto error_return;
402         end;
403
404 /* DSC-18 to DSC-20: See if dtu or dtm of these directories has been modified.
405       Normally, the dtu and dtm of (18) would be modified, and the dtu of
406       (19) and (20), but the access classes of their parents are below our authorization. */
407
408      do i = 18 to 20;
409           call set_saved_loc ( 409, "DSC-" || number_((i)));
410           call hcs_$status_ ((pathnames(i)), (enames(i)), 0, addr(test_status), null(), code);
411           if code ^= 0 then do;
412                call com_err_ (code, who, "^a>^a");
413                call ioa_ ("^a: The status information for ^a>^a
414 was available before the tests, but not afterwards.
415 Test ^a could not be made.  Testing continues.",
416                          who, pathnames(i), enames(i), saved_name);
417                end;
418           else do;
419                if saved_status(i).dtm ^= test_status.dtm then call bad_status ("date-time modified", i);
420                if saved_status(i).dtu ^= test_status.dtu then call bad_status ("date-time used", i);
421                end;
422      end;
423
424      call cleanup_proc;
425 end;
426 real_return:
427
428 return;
429
430 error_return:
431
432    call ioa_ ("Test was ^a: Error occurred on line ^d of ^a.", saved_name, saved_loc, who);
433    call com_err_ (0, who, "Test failed.");
434 /*
```

```
*/
/* Procedure to print an error message with a status code */

code_error: proc (mode, code, pname, expected);
 dcl (mode, expected, pname) char(*);
 dcl quotes char(1);
 dcl code fixed bin(35);

 if expected = "no error." then quotes = ""; else quotes = """";
 call ioa_ ("Segment ^a>^a^/gave error ""^a""^/on ^a instead of ^a^a^a",
          dirname, pname, convert_status_code_ (code, ""), mode, quotes, expected, quotes);
 goto error_return;
end;


/* Procedure to print an error message with a condition name */

condition_error: proc (mode, condition, pname, expected);
 dcl (mode, condition, pname, expected) char(*);
 dcl quotes char(1);
 if expected = "none" then quotes = ""; else quotes = """";
 call ioa_ ("Condition ""^a"" occurred on ^a^/of segment ^a>^a^/instead of ^a^a^a.",
          condition, mode, dirname, pname, quotes, expected, quotes);
 goto error_return;
end;

/* Procedure to create an upgraded directory and check status codes */

upgrade: proc (dir, class, quota, expected_code);
 dcl dir char(32) varying;
 dcl class bit(72) aligned;
 dcl expected_code fixed bin(35);
 dcl code fixed bin(35);
 dcl quota fixed bin(18);

 branch_.quota = quota;
 branch_.access_class = class;
 if dir = ""     /* null directory name means [pd] */
 then call hcs_$create_branch_ (get_pdir_(), "upgraded_dir", addr(branch_), code);
 else call hcs_$create_branch_ (dirpath || ">" || dir, "upgraded_dir", addr(branch_), code);
 if code ^= expected_code then do;
     call convert_authorization_$to_string (class, s1, 0);
     if code ^= 0
     then call ioa_$nnl ("^/Status code ""^a""^/returned on create of directory ",
                    convert_status_code_ (code, ""));
     else call ioa_$nnl ("^/No status code returned on create of directory ");
     if dir = ""
     then call ioa_ ("[pd]>upgraded_dir");
     else call ioa_ ("^a>^a>upgraded_dir", dirname, dir);
     call ioa_ ("(with access class ""^a"", quota = ^d)", s1, quota);
     if expected_code = 0
     then call ioa_$nnl ("instead of no code.  ");
     else call ioa_ ("instead of ""^a""", convert_status_code_ (expected_code, ""));
     call ioa_ ("Reference was by hcs_$create_branch_.");
     goto error_return;
     end;
end;
```

```
/* cleanup procedure for directory upgrade tests */

cleanup_proc: proc;
 dcl i fixed bin;

 do i = 1 to 6; /* delete any extraneous directories that might exist */
   call hcs_$delentry_file (dirpath || ">" || dirs_to_try(i), "upgraded_dir", 0);
 end;
 call hcs_$delentry_file (get_pdir_(), "upgraded_dir", 0);
end;

/* Procedure to save the line number and test name */

set_saved_loc: proc (loc, name);
 dcl loc fixed bin;
 dcl name char(*);
 saved_loc = loc;
 saved_name = name;
end;

/* Procedure to print a message for OSC-18 through OSC-20 */

bad_status: proc (name, number);
      dcl name char(*);
      dcl number fixed bin;
      call ioa_ ("^/The ^a returned in the status for^/^a>^a has been modified by these tests.",
                 name, pathnames(number), enames(number));
      goto error_return;
end;


/* Called when one onf the six directories was of the wrong access class */

bad_dir: proc (i);
      dcl i fixed bin;
      call convert_authorization_$to_string (class_bits, s1, code);
      call com_err_ (0, who, "Current authorization does not bear the proper relationship to the access class^/of the directory ^a>
^a
(^a).^/Test could not be run.", dirname, dirs_to_try(i), s1);
      goto real_return;
end;

end;
```

```
                    test_ipc.pl1                          07/29/75  1342.9 edt Tue                    page 1
/*  When this command is issued with six (6) arguments,
       the six args should correspond to:
           C,c1,c2  S,c1  S,c1,c2  S,c1,c2,c3  T,c1,c2  S,c1,c3
*/
test_ipc: tipc: proc;

   dcl   arg_1_len  fixed bin;
   dcl   arg_2_len  fixed bin;
   dcl   arg_3_len  fixed bin;
   dcl   arg_4_len  fixed bin;
   dcl   arg_5_len  fixed bin;
   dcl   arg_6_len  fixed bin;
   dcl   auth_1  char(arg_1_len) based(ptr_arg_1);
   dcl   auth_2  char(arg_2_len) based (ptr_arg_2);
   dcl   auth_3  char(arg_3_len) based(ptr_arg_3);
   dcl   auth_4  char(arg_4_len) based(ptr_arg_4);
   dcl   auth_5  char(arg_5_len) based(ptr_arg_5);
   dcl   auth_6  char(arg_6_len) based(ptr_arg_6);
   dcl   code  fixed bin(35);
   dcl   com_err_  entry options(variable);
   dcl   convert_authorization_$from_string  entry (bit(72) aligned,
                                char(*), fixed bin(35));
   dcl   cu_$arg_count  entry (fixed bin);
   dcl   cu_$arg_ptr  entry(fixed bin, ptr, fixed bin, fixed bin(35));
   dcl   hcs_$get_authorization  entry (bit(72) aligned, bit(72) aligned);
   dcl   ioa_  entry options(variable);
   dcl   num_args  fixed bin;
   dcl   present_authorization_bit  bit(72) aligned;
   dcl   present_max_auth_bit  bit(72) aligned;
   dcl   ptr_arg_1  ptr;
   dcl   ptr_arg_2  ptr;
   dcl   ptr_arg_3  ptr;
   dcl   (ptr_arg_4 , ptr_arg_5, ptr_arg_6)  ptr;
   dcl   ptr_single_arg  ptr;
   dcl   response_to_start_up  entry;
   dcl   single_arg  char(single_arg_len) based(ptr_single_arg);
   dcl   single_arg_len  fixed bin;
   dcl   system_low_bit  bit(72) aligned;
   dcl   terminal_2_proc  entry;
   dcl   tipc_set_up  entry (char(*), char(*), char(*), char(*), char(*),
                          char(*) );

  call   cu_$arg_count (num_args);
  if (num_args = 1)
     then do;
            /* Check that single arg = "-go"
            */
            call   cu_$arg_ptr (1, ptr_single_arg, single_arg_len, code);
            if (code ^= 0)
                 then do;
                         call   com_err_ (code, "tipc", "^/Could ^a",
                            "not find the single argument.");
                         return;
                    end;
            if (single_arg ^= "-go")
                 then do;
                         call   com_err_ (0, "tipc", "Single arg was not ""-go"".");
                         return;
```

```
                      end;

           /* The single arg is correct !
           */

          call   response_to_start_up;

          /* Since r_t_s_u does a new proc in all cases except when
             we are at system_low, we will return normally only then.
             We will also return here if r_t_s_u was in error and
             printed that error.
          */
          return;

        end;
if (num_args = 0)
  then do;
        /* Check that we not at system low....
        */
        call   hcs_$get_authorization (present_authorization_bit,
                            present_max_auth_bit);
        call   convert_authorization_$from_string (system_low_bit,
                                  "system_low", code);
        if (present_authorization_bit = system_low_bit)
          then co;
               call   com_err_ (0, "tipc",
                      "Not logged in at the correct authorization.");
               return;
             end;

        call   terminal_2_proc;

        /* We will normally return here!   In any case, terminal_2_proc
           will print its own messages.
        */
        return;
       end;

if (num_args = 6)
 then do;
         call   cu_$arg_ptr (1, ptr_arg_1, arg_1_len, code);
         if (code ^= 0)
          then do;
                 call   com_err_ (code, "tipc", "^/Could not find ^a",
                                  "first authorization arg.");
                 return;
               end;
         call   cu_$arg_ptr (2, ptr_arg_2, arg_2_len, code);
         if (code ^= 0)
          then do;
                 call   com_err_ (code, "tipc", "^/Could not find ^a",
                             "second authorization arg.");
                 return;
               end;
         call   cu_$arg_ptr (3, ptr_arg_3, arg_3_len, code);
         if (code ^= 0)
          then do;
                 call   com_err_ (code, "tipc", "^/Could not find ^a",
                                  "third authorization arg.");
                  return;
                end;
          call   cu_$arg_ptr (4, ptr_arg_4, arg_4_len, code);
          if (code ^= 0)
           then do;
                  call   com_err_ (code, "tipc", "^/Could not find ^a",
                                 "fourth authorization arg.");
                  return;
                end;
          call   cu_$arg_ptr (5, ptr_arg_5, arg_5_len, code);
          if (code ^= 0)
           then do;
                  call   com_err_ (code, "tipc", "^/Could not find ^a",
                                 "fifth authorization arg.");
                  return;
                end;
          call   cu_$arg_ptr (6, ptr_arg_6, arg_6_len, code);
          if (code ^= 0)
           then do;
                  call   com_err_ (code, "tipc", "^/Could not find ^a",
                                 "sixth authorization arg.");
                  return;
                end;

          /* Check that we ARE at system low...
          */
          call   hcs_$get_authorization (present_authorization_bit,
                               present_max_auth_bit);
          call   convert_authorization_$from_string (system_low_bit,
                                  "system_low", code);
          if (present_authorization_bit ^= system_low_bit)
           then do;
                  call   com_err_ (0, "tipc",
                           "Must be logged in at ""system_low"".");
                  return;
                end;


          call   tipc_set_up(auth_1, auth_2, auth_3, auth_4, auth_5,
                          auth_6);

          /* We should never return here, since tipc_set_up does a
             a new_proc_.  If we do, then tipc_set_up will have
             printed its error.
          */
          return;
        end;

   /* If we get here then num_args ^= 6,0,11  thus error!
   */
   call   com_err_ (0, "tipc", "Called with incorrect number (^2d) of arguments.", num_args);
   return;

end;
```

```
tlpc_set_up: proc (auth_1, auth_2, auth_3, auth_4, auth_5, auth_6);

   dcl   01 acl_addition(1) alignec,
           02 name char(32)  init("*.*.*"),
           02 modes bit(36)  init("1"b),
           02 pad bit(36)  init( (36)"0"b),
           02 code fixed bin(35);
   dcl   addr  builtin;
   dcl   before  builtin;
   dcl   cleanup  condition;
   dcl   (auth_1, auth_2, auth_3, auth_4, auth_5, auth_6) char(*);
   dcl   code  fixed bin(35);
   dcl   comm_with_2  char(3) init("no");
   dcl   com_err_  entry options(variable);
   dcl   convert_authorization_$from_string  entry (bit(72) aligned,
                                             char(*), fixed bin(35));
   dcl   cv_oct_check_  entry (char(*), fixed bin(35)) returns(fixed bin(35));
   dcl   entry_name  char(32) init("multi_process_info");
   dcl   error_table_$namedup  fixed bin(35) external;
   dcl   first_char  char(1);
   dcl   hcs_$add_acl_entries  entry (char(*), char(*), ptr, fixed bin,
                              fixed bin(35));
   dcl   hcs_$make_seg  entry (char(*), char(*), char(*), fixed bin(5),
                        ptr, fixed bin(35));
   dcl   hcs_$wakeup  entry (bit(36), fixed bin(71), fixed bin(71), fixec bin(35));
   dcl   hdir  char(68);
   dcl   01 info  aligned based(ptr_info),
          02 process_2_id  bit(36),
          02 channel_id  fixed bin(71),
          02 authorization_1 bit(72)aligned,
          02 authorization_2 bit(72)aligned,
          02 authorization_3 bit(72)aligned,
          02 authorization_4 bit(72)aligned,
          02 authorization_5 bit(72)aligned,
          02 authorization_6 bit(72)aligned;
   dcl   01 info_alternative   aligned based (ptr_info),
          02 process_2_id_bin  fixed bin(35),
          02 not_even_word_fill  fixed bin(35),
          02 first_half_chan_id fixed bin(35),
          02 second_half_chan_id fixed bin(35),
          02 x1 bit(72) aligned,
          02 x2 bit(72) aligned,
          02 x3 bit(72) aligned,
          02 x4 bit(72) aligned,
          02 x5 bit(72) aligned,
          02 x6 bit(72) aligned;
   dcl   ioa_  entry options(variable);
   dcl   ioa_$nnl  entry options(variable);
   dcl   ios_$read_ptr  entry ( ptr, fixec bin, fixec bin);
   dcl   ios_$resetread  entry (char(*), bit(72) aligned);
   dcl   message  fixed bin(71);
   dcl   new_proc_  entry (bit(72) aligned, fixed bin(35));
   dcl   null  builtin;
   dcl   num_chars  fixed bin;
   dcl   number_response  char(48);
   dcl   path_name_info_seg  char(168);
   dcl   process_2_exists  char(3) init("no");
   dcl   ptr_acl_addition  ptr;
dcl   ptr_info  ptr  init(null);
dcl   ptr_number_response  ptr;
dcl   ptr_response  ptr;
dcl   response  char(132);
dcl   rw_mode  fixed bin(5) init(1010b);
dcl   status  bit (72) aligned;
dcl   substr  builtin;
dcl   user_info_$homedir  entry (char(*));



/* Create info seg in your homedir.
*/
call   user_info_$homedir (hdir);


path_name_info_seg = before (hdir, " ") || ">" || entry_name;


on  cleanup  call   tipc_set_up_clean;
call   hcs_$make_seg (hdir, entry_name, "", rw_mode, ptr_info, code);
if ((code ^= 0) & (code ^= error_table_$namedup))
 then do;
        call   com_err_ (code, "tipc", "^/Could not create segment  ^a",
                          path_name_info_seg);
        call   tipc_set_up_clean;
        return;
      end;


/* Give *.*.* r_access to this info segment in your hdir.
*/
ptr_acl_addition = addr (acl_addition);
call   hcs_$add_acl_entries (hdir, entry_name, ptr_acl_addition,
                            1, code);
if (code ^= 0)
  then do;
        call   com_err_ (code, "tipc", "^/Could not give *.*.* ^a ^/^-^a",
                         "r_access to the segment!", path_name_info_seg);
        call   tipc_set_up_clean;
        return;
      end;



/* Store the six authorization arguments into info segment.
*/
call   convert_authorization_$from_string (authorization_1, autr_1, code);
if (code ^= 0)
 then do;
        call   com_err_ (code, "tipc", "^/Could not convert ^a",
                         "first authorization arg to bit str form.");
        call   tipc_set_up_clean;
        return;
      end;
call   convert_authorization_$from_string (authorization_2, autr_2, code);
if (code ^= 0)
  then do;
```

```
          call   com_err_ (code, "tipc", "^/Could not convert ^a",
                         "second authorization arg to bit str form.");
          call   tipc_set_up_clean;
          return;
        end;
call   convert_authorization_$from_string (authorization_3, auth_3, code);
if (code ^= 0)
  then do;
          call   com_err_ (code, "tipc", "^/Could not convert ^a",
                         "third authorization arg to bit str form.");
          call   tipc_set_up_clean;
          return;
        end;
call   convert_authorization_$from_string (authorization_4, auth_4, code);
if (code ^= 0)
  then do;
          call   com_err_ (code , "tipc", "^/Could not convert ^a",
                         "fourth authorization arg to bit str form.");
          call   tipc_set_up_clean;
          return;
        end;
call   convert_authorization_$from_string (authorization_5, auth_5 , code);
if (code ^= 0)
   then do;
          call   com_err_ (code , "tipc", "^/Could not convert ^a",
                         "fifth authorization arg to bit str form");
          call   tipc_set_up_clean;
          return;
         end;
call   convert_authorization_$from_string (authorization_6, auth_6, code);
if (code ^= 0)
 then do;
           call   com_err_ (code, "tipc", "^/Could not convert ^a",
                          "sixth authorization arg to bit str form.");
           call   tipc_set_up_clean;
           return;
          end;



/* Get user to goto second terminal and login at authorization_3.
*/
call   ioa_ ("*** Login at second terminal ^a ^a. ^/^a",
             "with authorization = ", auth_3,
             "*** Issue the command   ""test_ipc"" .");
ptr_response = addr (response);

at   call   ios_$resetread ("user_input", status);
   call   ios_$read_ptr (ptr_response, 132, num_chars);
   first_char = substr (response, 1, 1);
   if (first_char = "f")
    then do;
           call   tipc_set_up_clean;
           return;
          end;
   if (first_char ^= "s")
    then do;
           call   ioa_ ("^/^-!!! Did you fail(f) or succeed(s) ?");
```

```
            goto  a;
        end;





/* Presumably, we logged in at second terminal.
     1. with authorization = auth_3
     2. Issued the cmd  "test_ipc"
     3. were instructed there to come back to this terminal and type an s.
   Using the output on terminal 2, answer the following.
*/
process_2_exists = "yes";

call   ioa_ ("^/^/*** Using the output from other terminal, ^a",
                "answer the following. ");
ptr_number_response = addr (number_response);
b: call   ioa_$nnl ("^-First number = ");
   call   ios_$resetread ("user_input", status);
   call   ios_$read_ptr (ptr_number_response, 48, num_chars);

   /* Recall that number_response likely includes a lf character.
   */
   process_2_id_bin = cv_oct_check_ (substr (number_response, 1, num_chars - 1), code);
   if (code ^= 0)
     then do;
            call   ioa_ ("*** Use only digits !");
            goto  b;
          end;

c: call   ioa_$nnl ("^-Second number = ");
   call   ios_$resetread ("user_input", status);
   call   ios_$read_ptr (ptr_number_response, 48, num_chars);
   first_half_chan_id = cv_oct_check_ (substr ( number_response, 1, num_chars - 1), code);
   if (code ^= 0)
    then do;
            call   ioa_ ("*** Use only digits !");
            goto  c;
          end;

d: call   ioa_$nnl ("^-Third number = ");
   call   ios_$resetread  ("user_input", status);
   call   ios_$read_ptr (ptr_number_response, 48, num_chars);
   second_half_chan_id = cv_oct_check_ (substr(number_response, 1, num_chars - 1), code);
   if (code ^= 0)
    then do;
            call   ioa_ ("*** Use only digits !");
            goto d;
         end;



/*  Test the input numbers with an initial wakeup of process on other terminal.
*/

message = -1;
call   hcs_$wakeup ((process_2_id), channel_id,  message , code);
```

```
if (code ^= 0)
      then do;
                  call    com_err_ (0, "tipc", "One of the three numbers was  ^a ^/^/^a",
                                "typed in incorrectly !", "*** AGAIN, answer the following.");

                  goto   b;
            end;
comm_with_2 = "yes";




 /* 1. Process_2 has been createc and is waiting.
    2. Info seg in homedir is filled.
    3. We can communicate with other process.(ie.. numbers were input correctly)

    Now new_proc_ to authorizaticn_1.
      This will 1. activate  start_up.ec
                     2. which will call "tipc" with one arg.

 */
 call    new_proc_ (authorization_1, coce);




 /* We snould NOT return here, urless new_proc_ failed!!!
 */
 call    com_err_ (code, "tipc", "^/Coulc not create a ^a ^a",
                    "new process with authorization = ", auth_1);

 call    tipc_set_up_clean;
 return;



  tipc_set_up_clean:  proc;
     dcl    hcs_$delentry_seg  entry (ptr, fixed bin(35));
     dcl    status_code  fixed bin(35);

     if (process_2_exists = "yes")
          then  if (comm_with_2 = "yes")
                    then  call    hcs_$wakeup ((process_2_id), channel_id, 7, status_code);
                    else  call    ioa_ ("^/*** QUIT and release on other terminal.");
     call    hcs_$delentry_seg (ptr_info, status_code);

   end;

end;
```

```
/* This procedure references a given directory using all the hcs_ calls documented in
   the MPM.  The calls requiring s permission are segregated from the calls requiring sm
   permission.  The caller specifies the effective access mode he expects to have on a
   given directory (s, sm, or null), the name of the directory, and whether this directory
   is expected to have a higher access class than the process authorization.
   The user must also have
   a dummy subdirectory and a dummy segment within this directory for this routine to play
   with.  This routine will print, on user_output, error messages in case of an incorrect
   status code, or in case it thinks access was allowed when it shouldn't have been.
   The quota of the directory to be tested should be at least 3 so that the quota_move
   primitive can be tested.

   The subdirecory provided within parent should be empty.
   The segment should contain all zero words, except for the first bit which should be "1"b.
   There should be no other entries in parent--if there are, this subroutine will work
   right but the other entries may be deleted (because hcs_$del_dir_tree is one of the tests).
   The following attributes of the directory and segment must be set:

                    directory segment
    bitcount            0         1
    quota               0
    rings             7,7      4,4,4
    safety switch                 0
    max length              1024 words
*/

try_dir_reference_: proc (parent, dirname, segname, mode, upgrade, error);
dcl (parent,                               /* directory to which access is to be tested */
     dirname,                              /* name of a subdirectory within parent */
     segname,                              /* name of a segment in parent */
     mode) char(*);                        /* mode expected on parent */
dcl upgrade bit(1);              /* "1"b access class of parent is not lower than or equal to process authorization */

/* Note: if the upgrade flag is set, mode should be "null", since there can be no effective access
   to anything within parent. */

/* The mode argument should be "", "n", "s", or "sm".
   Status permission to parent is tested by reading the attributes of dirname
   and segname.  Modify permission is tested by performing file system operations within parent
   such as trying to create segments and directories, setting parent's initial ACL, etc.
   The branch of parent is never referenced, since access to it is controlled by the ACL
   of its parent.  Dirname and segname should always be of the same access class
   as the parent and full access (sma and rew) should be given--otherwise there will be errors
   all over the place.

   This program must know exactly which error_table_ codes are returned by hcs_ calls
   in different situations.  The status codes expected are stored in the variables
   allowed_code, and not_allowed_code, depending on whether the particular
   access mode being tested was allowed or not.  In most legitimate cases, allowed_code
   will be zero, indicating no status code expected when access is allowed.  However,
   tests are made with directory-referencing hcs_ calls that are passed the pathname of
   a segment, and vice versa (for example hcs_$list_dir_acl of a segment).  In these
   cases, access may have been allowed except that the entry is of the
   wrong type.  The value of allowed_code, therefore, is nonzero.  The value of not_allowed_code
   usually depends on whether the directory being referenced is of a higher access class than
   the parent.  That's the reason for the upgrade argument.

   If a system change results in a status code being returned that is not the one
```

```
expected (in some particular case), then a change must be made to this program as follows.

1. Run the program again, but first call check_status_$debug_on and check_status_$return
   so that all error messages will appear.  The following steps should be performed
   for every status code that appears to be wrong, although only one small change in the program
   is usually necessary to correct a large number of bad status codes.

2. Locate the section of code in this program that references the hcs_ entry,
   using the line number supplied in the error message printed.
   Find the call to check_status_$set that precedes this hcs_ call,
   usually at the top of one of the preceding pages.  The first
   argument to check_status_$set is either "s" for Status or "m" for Modify,
   indicating the mode begin tested.

3. Determine (from the error message) whether the reference was to a directory or
   segment of a higher, equal, or lower access class than the process authorization.
   This determines the "effective mode" of access the the entry.  That is,
   if access class is higher, effective mode is null, if access class is lower,
   effective mode is s, and if access class is equal effective mode is s and m.

4. If the mode being tested (as determined in step 2) is included in the
   effective mode determined in step 3, then the value of "allowed_code" must
   be changed to conform the the error_table_ code that was actually returned.
   If the mode being tested is not in the effective mode, the value of "not_allowed_code"
   must be changed.  If "not_allowed_code" is to be changed, make sure it is changed
   only in the case when "upgrade" or "^upgrade" is specified, as determined by whether the
   directory referenced is within or outside the authorization of the process.

5. After making the change to this program, run the source through
   line_number_inserter to update the line numbers in the check_status_,
   list_acl_test and set_acl_test calls, and then recompile this procedure.
   */

dcl error fixed bin(35);
dcl change_wdir_ entry (char(168) aligned, fixed bin(35));
dcl convert_authorization_$to_string entry (bit(72) aligned, char(*), fixed bin(35));
dcl convert_status_code_ entry (fixed bin(35), char(8) aligned) returns (char(100) aligned);
dcl date_time_$fstime entry (fixed bin(35), char(*));
dcl expand_path_ entry (ptr, fixed bin, ptr, ptr, fixed bin(35));
dcl get_group_id_$tag_star entry returns (char(32) aligned);
dcl get_pdir_ entry returns (char(168) aligned);
dcl get_ring_ entry returns (fixed bin(6));
dcl get_system_free_area_ entry returns (ptr);
dcl get_wdir_ entry returns (char(168) aligned);
dcl hcs_$add_acl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin(35));
dcl hcs_$add_dir_acl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin(35));
dcl hcs_$add_dir_inacl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$add_inacl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$append_branch entry (char(*), char(*), fixed bin(5), fixed bin(35));
dcl hcs_$create_branch_ entry (char(*), char(*), ptr, fixed bin(35));
dcl hcs_$append_branchx entry (char(*), char(*), fixed bin(5), (3) fixed bin(3), char(*),
        fixed bin(1), fixed bin(1), fixed bin(24), fixed bin(35));
dcl hcs_$append_link entry (char(*), char(*), char(*), fixed bin(35));
dcl hcs_$chname_file entry (char(*), char(*), char(*), char(*), fixed bin(35));
dcl hcs_$chname_seg entry (ptr, char(*), char(*), fixed bin(35));
dcl hcs_$del_dir_tree entry (char(*), char(*), fixed bin(35));
dcl hcs_$delentry_file entry (char(*), char(*), fixed bin(35));
dcl hcs_$delentry_seg entry (ptr, fixed bin(35));
dcl hcs_$delete_acl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin(35));
dcl hcs_$delete_dir_acl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin(35));
dcl hcs_$delete_dir_inacl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$delete_inacl_entries entry (char(*), char(*), ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$fs_move_file entry (char(*), char(*), fixed bin(2), char(*), char(*), fixed bin(35));
dcl hcs_$fs_move_seg entry (ptr, ptr, fixed bin(1), fixed bin(35));
dcl hcs_$fs_search_set_wdir entry (char(*) aligned, fixed bin(35));
dcl hcs_$get_access_class entry (char(*), char(*), bit(72) aligned, fixed bin(35));
dcl hcs_$get_author entry (char(*), char(*), fixed bin(1), char(*), fixed bin(35));
dcl hcs_$get_authorization entry (bit(72) aligned, bit(72) aligned);
dcl hcs_$get_bc_author entry (char(*), char(*), char(*), fixed bin(35));
dcl hcs_$get_dir_ring_brackets entry (char(*), char(*), (2) fixed bin(3), fixed bin(35));
dcl hcs_$get_max_length entry (char(*), char(*), fixed bin(19), fixed bin(35));
dcl hcs_$get_ring_brackets entry (char(*), char(*), (3) fixed bin(3), fixed bin(35));
dcl hcs_$get_safety_sw entry (char(*), char(*), bit(1), fixed bin(35));
dcl hcs_$get_search_rules entry (ptr);
dcl hcs_$initiate entry (char(*), char(*), char(*), fixed bin(1), fixed bin(2), ptr, fixed bin(35));
dcl hcs_$initiate_count entry (char(*), char(*), char(*), fixed bin(24),
        fixed bin(2), ptr, fixed bin(35));
dcl hcs_$initiate_search_rules entry (ptr, fixed bin(35));
dcl hcs_$list_acl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin(35));
dcl hcs_$list_dir_acl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin(35));
dcl hcs_$list_dir_inacl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$list_inacl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin, fixed bin(35));
dcl hcs_$make_ptr entry (ptr, char(*), char(*), ptr, fixed bin(35));
dcl hcs_$make_seg entry (char(*), char(*), char(*), fixed bin(5), ptr, fixed bin(35));
dcl hcs_$quota_get entry (char(*), fixed bin(18), fixed bin(35), bit(36) aligned, fixed bin,
        fixed bin(1), fixed bin, fixed bin(35));
dcl hcs_$quota_move entry (char(*), char(*), fixed bin(18), fixed bin(35));
dcl hcs_$replace_acl entry (char(*), char(*), ptr, fixed bin, bit(1), fixed bin(35));
dcl hcs_$replace_dir_acl entry (char(*), char(*), ptr, fixed bin, bit(1), fixed bin(35));
dcl hcs_$replace_dir_inacl entry (char(*), char(*), ptr, fixed bin,
        bit(1) aligned, fixed bin, fixed bin(35));
dcl hcs_$replace_inacl entry (char(*), char(*), ptr, fixed bin, bit(1), fixed bin, fixed bin(35));
dcl hcs_$set_bc entry (char(*), char(*), fixed bin(24), fixed bin(35));
dcl hcs_$set_dir_ring_brackets entry (char(*), char(*), (2) fixed bin(3), fixed bin(35));
dcl hcs_$set_bc_seg entry (ptr, fixed bin(24), fixed bin(35));
dcl hcs_$set_max_length entry (char(*), char(*), fixed bin(19), fixed bin(35));
dcl hcs_$set_ring_brackets entry (char(*), char(*), (3) fixed bin(3), fixed bin(35));
dcl hcs_$set_safety_sw entry (char(*), char(*), bit(1), fixed bin(35));
dcl hcs_$set_safety_sw_seg entry (ptr, bit(1), fixed bin(35));
dcl hcs_$star_ entry (char(*), char(*), fixed bin(2), ptr, fixed bin, ptr, ptr, fixed bin(35));
dcl hcs_$star_list_ entry (char(*), char(*), fixed bin(3), ptr, fixed bin, fixed bin,
        ptr, ptr, fixed bin(35));
dcl hcs_$status_ entry (char(*), char(*), fixed bin(1), ptr, ptr, fixed bin(35));
dcl hcs_$status_long entry (char(*), char(*), fixed bin(1), ptr, ptr, fixed bin(35));
dcl hcs_$status_mint entry (char(*), char(*), fixed bin(1), fixed bin(2), fixed bin(24),
        fixed bin(35));
dcl hcs_$terminate_file entry (char(*), char(*), fixed bin(1), fixed bin(35));
dcl hcs_$terminate_name entry (char(*), fixed bin(35));
dcl hcs_$truncate_file entry (char(*), char(*), fixed bin, fixed bin(35));
dcl hcs_$truncate_seg entry (ptr, fixed bin, fixed bin(35));
dcl code fixed bin(35);
dcl error_table_$argerr external fixed bin(35);
dcl error_table_$bad_ring_brackets external fixed bin(35);
dcl error_table_$dirseg external fixed bin(35);
dcl error_table_$incorrect_access external fixed bin(35);
dcl error_table_$moderr external fixed bin(35);
dcl error_table_$name_not_found external fixed bin(35);
dcl error_table_$namedup external fixed bin(35);
dcl error_table_$no_info external fixed bin(35);
dcl error_table_$no_linkage external fixed bin(35);
dcl error_table_$no_move external fixed bin(35);
dcl error_table_$no_s_permission external fixed bin(35);
dcl error_table_$nondirseg external fixed bin(35);
dcl error_table_$notadir external fixed bin(35);
dcl error_table_$segknown external fixed bin(35);
dcl error_table_$user_not_found external fixed bin(35);

dcl dummy_code fixed bin(35); /* just storage */
dcl not_allowed_code fixed bin(35); /* expected code when mode tested was not supposed to be allowed */
                                /* The value of expected code depends on whether the access class of the
                                   directory is greater than the process authorization or not.
                                   i.e., whether "upgrade" is set. */
dcl allowed_code fixed bin(35);/* expected code when mode tested was supposed to be allowed */
dcl access_class bit(72) aligned;
dcl authorization bit(72) aligned;
dcl pd char(168) init (get_pdir_());
dcl wdir char(168) aligned init ("");
dcl pathname char(168);                 /* storage for a pathname */
dcl scratch char(32);                   /* scratch storage */
dcl (dir_path, seg_path, x_path) char(168);
dcl (dir_pathx, seg_pathx) char(168) varying;
dcl mode_expected bit(2) init("00"b);   /* access mode the caller expected to have */
dcl reference char(168);                /* actual pathname that is being referenced for a particular test */
dcl userid char(32);                    /* This userid */
dcl parent_parent char(168);            /* parent of parent */
dcl parent_ename char(32);              /* entry name of parent */
dcl tempname char(32) init ("dir_ref_temp");
dcl tempptr ptr init(null);
dcl segptr ptr;
dcl cleanup condition;
dcl null builtin;
dcl first_bit bit(1) based;
dcl (s_expected, m_expected) bit(1) init("0"b); /* set depending on mode_expected */

/* codes representing s or m permission, used as literal constants in various places */

dcl (s init ("01"b),
     m init ("10"b),
     n init ("00"b)) bit(2) static;

/* Special routines */

dcl check_status_$set entry (bit(2), bit(2), ptr, ptr, ptr, label, ptr);
dcl check_status_ entry options(variable);

dcl conv_$fb entry (fixed bin(35)) returns (char(20));
dcl conv_$ptr entry (ptr) returns (char(20));
dcl conv generic (conv_$fb when (fixed bin(1:35)), conv_$ptr when (ptr));

dcl areap ptr;

areap = get_system_free_area_();
x_path = before (parent, " ") || ">x";
seg_path = before (parent, " ") || ">" || segname;
```

```
dir_path = before (parent, " ") || ">" || dirname;
dir_pathx = before (dir_path, " ");
seg_pathx = before (seg_path, " ");
call expand_path_ (addr(parent), length(parent), addr(parent_parent), addr(parent_ename), code);
if code ^= 0 then do;
   error = code;                    /* something bad about parent pathname */
   return;
   end;
userid = get_group_id_$tag_star ();
wdir = get_wdir_();
ring = get_ring_();

/* Determine if this is a system with access isolation, and if it is,
   set a flag. */

dcl ai bit(1) init ("1"b);
dcl linkage_error condition;

on linkage_error begin;
     ai = "0"b;
     goto no_ai;
     end;
call hcs_$get_authorization (authorization, "0"b);
no_ai:
revert linkage_error;

/* Set flags for access mode we think we have */

if index (mode, "m") ^= 0 then do;
     mode_expected = m;
     m_expected = "1"b;
     end;
if index (mode, "s") ^= 0 then do;
     mode_expected = mode_expected | s;
     s_expected = "1"b;
     end;
/* if mode = "n" | mode = "", no bits in mode_expected get set */

/* Now establish a cleanup on unit */

on cleanup call cleanup_stuff;

/* initialize stuff in structure for hcs_$create_branch_  */

%include create_branch_info;
dcl 1 branch_ like create_branch_info aligned;

branch_.version = create_branch_version_1;
branch_.switches = ""b;
branch_.dir_sw = "1"b;
branch_.mode = "111"b;
branch_.mbz2 = ""b;
branch_.rings = ring;
branch_.userid = "xxxxx.xxxxx.x";
branch_.bitcnt, branch_.quota = 0;
branch_.access_class = authorization;

/* Make a temporary segment in process dir with which to play.
```

```
   This segment is used mostly for the move primitives */

call ncs_$make_seg (pd, tempname, "", 01110b, temptr, code);
if temptr = null then do;
     error = code;
     return;
     end;
/*
```

```
*/

/* Before trying any tests, save certain information that is needed to restore in
   case of cleanup and at end of tests. */

dcl 1 saved_search_rules,
      2 num fixed bin init(0),
      2 names (21) char(168) aligned;

dcl 1 segment_acl(1),
      2 name char(32) init ("xxxxx.xxxxx.x"),
      2 modes bit(36) init (""b),
      2 pad bit(36) init (""b),
      2 code fixed bin(35);

dcl 1 saved_seg_acl (saved_seg_acl_count) based (saved_seg_acl_ptr) like segment_acl;
dcl 1 dir_acl(1),
      2 name char(32) init ("xxxxx.xxxxx.x"),
      2 modes bit(36) init (""b),
      2 code fixed bin(35);

dcl 1 saved_dir_acl(saved_dir_acl_count) based (saved_dir_acl_ptr) like dir_acl;

dcl 1 saved_dir_inacl (saved_dir_inacl_count) based (saved_dir_inacl_ptr) like dir_acl;
dcl 1 saved_seg_inacl (saved_seg_inacl_count) based (saved_seg_inacl_ptr) like segment_acl;

dcl (saved_seg_acl_count, saved_dir_acl_count, saved_seg_inacl_count, saved_dir_inacl_count) fixed bin;
dcl (saved_dir_acl_ptr, saved_seg_acl_ptr, saved_seg_inacl_ptr, saved_dir_inacl_ptr) ptr init (null);

dcl 1 delete_acl(1),
      2 name char(32) init ("yyyyy.yyyyy.y"),
      2 code fixed bin(35);

dcl ring fixed bin;

dcl saved_quota fixed bin(18) init(0); /* saved quota of parent */

/* We are saving the current ring, search rules, and working directory.
   The ACL of dirname and segname is saved, and the initial ACLs in parent are saved */

call hcs_$get_search_rules (addr(saved_search_rules));
call hcs_$list_acl (parent, segname, areap, saved_seg_acl_ptr, null, saved_seg_acl_count, code);
call hcs_$list_dir_acl (parent, dirname, areap, saved_dir_acl_ptr, null, saved_dir_acl_count, code);
call hcs_$list_inacl (parent_parent, parent_ename, areap, saved_seg_inacl_ptr, null, saved_seg_inacl_count, ring, code);
call hcs_$list_dir_inacl (parent_parent, parent_ename, areap, saved_dir_inacl_ptr, null, saved_dir_inacl_count, ring, code);

/* Save the quota of parent, because hcs_$del_dir_tree changes it. */

call hcs_$quota_get (parent, saved_quota, 0, ""b, 0, 0, 0, 0);

/* In case there was no access and these ACL structures weren't allocated,
   point them to dummies */

call fix (saved_seg_acl_ptr, saved_seg_acl_count, addr(segment_acl));
call fix (saved_dir_acl_ptr, saved_dir_acl_count, addr(dir_acl));
call fix (saved_seg_inacl_ptr, saved_seg_inacl_count, addr(segment_acl));
call fix (saved_dir_inacl_ptr, saved_dir_inacl_count, addr(dir_acl));

```

```
fix: proc (ptr, count, dummy_ptr);
 dcl ptr ptr;
 dcl count fixed bin;
 dcl dummy_ptr ptr;
 if ptr = null then do;
    ptr = dummy_ptr;
    count = 1;
    end;
end;
/*
```

```
*/
366 /* Test legitimate directory references that require only s permission to parent. */
367
368 /* Initialize check_status_ routine */
369
370 if upgrade
371 then not_allowed_code = error_table_$incorrect_access;
372 else not_allowed_code = error_table_$incorrect_access;
373 call check_status_$set (s, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
374 reference = dir_path;                        /* we will be referencing dir_path at first */
375 allowed_code = 0;
376
377 /* Try to list the ACL of dir_path and its initial ACLs */
378
379 call list_acl_test ( 379, 37, hcs_$list_dir_acl, parent, dirname);
380 call list_acl_test ( 380, 39, hcs_$list_inacl, parent_parent, parent_ename);
381 call list_acl_test ( 381, 38, hcs_$list_dir_inacl, parent_parent, parent_ename);
382
383 /* Try to get author */
384
385 scratch = "";
386 call hcs_$get_author (parent, dirname, 0, scratch, code);
387 call check_status_ ( 387, 25, scratch ^= "", "author's name returned: ^a", scratch);
388 scratch = "";
389 call hcs_$get_bc_author (parent, dirname, scratch, code);
390 call check_status_ ( 390, 26, scratch ^= "", "author's name returned: ^a", scratch);
391
392 /* Try to get ring brackets */
393
394 dcl rb(3) fixed bin(3);
395 dcl drb(2) fixed bin(3) based (addr(rb(1)));
396 drb = -1;
397 call hcs_$get_dir_ring_brackets (parent, dirname, drb, code);
398 call check_status_ ( 398, 27, sum(drb) ^=  -2, "ring brackets (^a, ^a) were returned", drb(1), drb(2));
399
400 /* Try to get the quota */
401
402 dcl quota fixed bin(18),
403      trp fixed bin(35),
404      tup bit(36) aligned,
405      infqcnt fixed bin,
406      taccsw fixed bin(1),
407      used fixed bin;
408 quota, trp, infqcnt, taccsw, used = -1;
409 tup = "0"b;
410 reference = parent;
411 call hcs_$quota_get (parent, quota, trp, tup, infqcnt, taccsw, used, code);
412 if tup ^= "" then call date_time_$fstime (fixed(tup, 35), scratch);
413              else scratch = "not returned";
414 call check_status_ ( 414, 42, quota ^= 0 | infqcnt+taccsw+used ^= 0 | tup ^= "0"b,
415      "some quota information was returned: quota = ^a, time-record-product = ^a,
416 time_updated ""^a"", inferior directories = ^a, terminal account switch = ^a,
417 records used ^a", conv((quota)), conv((trp)), scratch, conv((infqcnt)), conv((taccsw)), conv((used)));
418
419 /* Try the status commands */
420
421 dcl branch bit(360) aligned; /* this is big enough to hold the longest status structure */
422 dcl result bit(1); /* set to "1"b if information was returned */
```

```
423
424 dcl 1 bbranch based(addr(branch)), /* overlay of status structure */
425        2 (type bit(2),
426           pad1 bit(106),       /* the pad fields are not returned when s permission is lacking */
427           mode bit(5),
428           pad2 bit(13),
429           records bit(18),     /* hcs_$status_ only returns up to here */
430           pad3 bit(108),
431           curlen bit(12),
432           bitcnt bit(24),
433           pad4 bit(72)) unaligned;
434
435 if ^upgrade then not_allowed_code = error_table_$no_s_permission;
436 branch=""b;
437 reference = dir_path;
438 call hcs_$status_ (parent, dirname, 0, addr(branch), areap, code);
439 if upgrade then result = branch ^= ""b; /* if upgrade, no information should be returned */
440            else result = (bbranch.pad1 | bbranch.pad2) ^= ""b;
441 call check_status_ ( 441, 59, result, "information about the branch was returned");
442
443 branch=""b;
444 call hcs_$status_long (parent, dirname, 0, addr(branch), areap, code);
445 if upgrade then result = branch ^= ""b;
446            else result = (bbranch.pad1 | bbranch.pad2 | bbranch.pad3 | bbranch.pad4) ^= ""b;
447 call check_status_ ( 447, 60, result, "information about the branch was returned");
448
449 /* Try to match something in parent with starname of ** */
450 /* This test is made with reference to the parent, since s permission on the
451    parent is required to call these entries */
452
453 if ^upgrade then not_allowed_code = error_table_$incorrect_access;
454 reference = before (parent, " ") || ">**";
455 dcl 1 entries (ecount) aligned based (eptr),
456       2 (type bit(2),
457          nnames bit(16),
458          nindex bit(18)) unaligned;
459 dcl names (1000) char(32) aligned based (nptr);
460 dcl (eptr, nptr) ptr init(null);
461 dcl ecount fixed bin;
462 ecount = -1;
463 eptr, nptr = null;
464 call hcs_$star_ (parent, "**", 3, areap, ecount, eptr, nptr, code);
465 call check_status_ ( 465, 57, ecount^=0 | eptr ^= null | nptr ^= null, "some information was returned.
466   count of entries=^a, pointer to data structure ^a, pointer to names ^a",
467   conv((ecount)), conv(eptr), conv(nptr));
468 if eptr ^= null then free entries;
469 if nptr ^= null then free names;
470
471 dcl (seg_count, link_count) fixed bin;
472 seg_count, link_count = -1;
473 eptr, nptr = null;
474 call hcs_$star_list_ (parent, "**", 7, areap, seg_count, link_count, eptr, nptr, code);
475 call check_status_ ( 475, 58, seg_count + link_count ^= 0 | nptr ^= null | eptr ^= null, "some information was returned");
476 if eptr ^= null then free entries;
477 if nptr ^= null then free names;
478
479 /* If access isolation is working, use hcs_$get_access_class on directory */
480
```

```
481 dcl access_class_name char(250);
482
483 if a1 then do;
484      reference = dir_path;
485      access_class = ""b;
486      call hcs_$get_access_class (parent, dirname, access_class, code);
487      call convert_authorization_$to_string (access_class, access_class_name, dummy_code);
488      if dummy_code ^= 0 then
489           access_class_name = convert_status_code_ (dummy_code, "xxxxxxxx");
490      call check_status_ ( 490, 71, access_class ^= ""b, "access class was returned. Its value was: ^a", access_class_name);
491      end;
492 /*
```

```
    */
493 /* Test directory references that require s to the directory, but none to the
494    parent.  It only makes sense to test these references when upgrade is
495    specified, because we know the ACL of dirname is sma. */
496
497 if upgrade then do;
498
499      /* Try to put dir_path in search rules.  We do this by adding dir_path
500         at end of current search rules.  The search rules are then read back,
501         and we see if they were changed. */
502
503      dcl 1 search_rules like saved_search_rules;
504
505      not_allowed_code = error_table_$incorrect_access;
506      call check_status_$set (s, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
507      search_rules = saved_search_rules;
508      search_rules.num = search_rules.num + 1;
509      search_rules.names(search_rules.num) = dir_path;
510      reference = dir_path || " as last search directory";
511      call hcs_$initiate_search_rules (addr(search_rules), code);
512      search_rules.names(search_rules.num) = "";   /* see if search rules were changed */
513      call hcs_$get_search_rules (addr(search_rules));
514      call check_status_ ( 514, 35, search_rules.names(saved_search_rules.num + 1) ^= "", " search rules were changed");
515      call hcs_$initiate_search_rules (addr(saved_search_rules), code); /* put back search rules
516                                                                         in case they went away */
517
518      /* Try to set wdir to dir_path */
519
520      reference = dir_path;
521      call hcs_$fs_search_set_wdir ((dir_path), code); /* set the working directory */
522      pathname = get_wdir_();   /* see if wdir was changed */
523      call check_status_ ( 523, 24, pathname ^= wdir, "working directory was set to ^a", pathname);
524      call change_wdir_ (wdir, code); /* change wdir back again in case it worked */
525
526      /* Try to get the safety switch */
527
528      reference = dir_path;
529      allowed_code = error_table_$dirseg;
530      dcl safety_sw bit(1) init("1"b);
531      allowed_code = 0;
532      call hcs_$get_safety_sw (parent, dirname, safety_sw, code);
533      call check_status_ ( 533, 31, safety_sw ^= "1"b, "safety switch was returned");
534
535      /* Try to get the max length of the directory */
536
537      dcl max_length fixed bin(19);
538      max_length = -1;
539      reference = dir_path;
540      call hcs_$get_max_length (parent, dirname, max_length, code);
541      call check_status_ ( 541, 28, max_length ^= -1, "max length returned! ^d", max_length);
542
543      dcl bc fixed bin(24);
544      dcl type fixed bin(2);
545      type, bc = -1;
546      call hcs_$status_mins (parent, dirname, 0, type, bc, code);
547      call check_status_ ( 547, 61, type+bc ^= -2, "type or bitcount was returned! type = ^a, bitcount = ^a",
548           conv((type)), conv((bc)));
549      end;
```

```
550 /*
```

```
*/
551 /* Test the directory references that require sm permission. */
552
553 not_allowed_code = error_table_$incorrect_access;
554
555 call check_status_$set (m, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), enc_all, addr(reference));
556 reference = dir_path;
557 allowed_code = 0;               /* if we have m permission, expect this code */
558
559 /* Try to modify the ACL of dir_path or initial ACLs of parent */
560
561 call set_acl_test ( 561, 2, parent, dirname);                              /* add_dir_acl_entries */
562 call set_acl_test ( 562, 10, parent, dirname);                   /* delete_dir_acl_entries */
563 call set_acl_test ( 563, 6, parent, dirname);                              /* replace_dir_acl */
564 reference = parent;
565 call set_acl_test ( 565, 4, parent_parent, parent_ename);    /* add_dir_inacl_entries */
566 call set_acl_test ( 566, 3, parent_parent, parent_ename);    /* add_inacl_entries */
567 call set_acl_test ( 567, 11, parent_parent, parent_ename);   /* delete_inacl_entries */
568 call set_acl_test ( 568, 12, parent_parent, parent_ename);   /* delete_dir_inacl_entries */
569 call set_acl_test ( 569, 7, parent_parent, parent_ename);    /* replace_inacl */
570 call set_acl_test ( 570, 8, parent_parent, parent_ename);    /* replace_dir_inacl */
571
572 /* Try to put a link or directory into parent using the append entries */
573
574 reference = x_path;
575 call hcs_$append_branchx (parent, "x", 0, ((ring)), "xxxxx.xxxxx.x", 1, 0, 0, code);
576 call hcs_$status_minf (parent, "x", 0, 0, 0, dummy_code);   /* see if it was created */
577 call check_status_ ( 577, 6, dummy_code = 0, "directory seems to have been created");
578 call hcs_$delentry_file (parent, "x", dummy_code);            /* delete directory if it was created */
579
580 if ai then do;
581      call hcs_$create_branch_ (parent, "x", addr(branch_), code);
582      call hcs_$status_minf (parent, "x", 0, 0, 0, dummy_code);
583      call check_status_ ( 583, 70, dummy_code = 0, "directory seems to have been created");
584      call hcs_$delentry_file (parent, "x", 0);
585      end;
586
587 call hcs_$append_link (parent, "x", ">x", code); /* link points to nowhere meaningful */
588 call hcs_$status_minf (parent, "x", 0, 0, 0, dummy_code);   /* see if it was created */
589 call check_status_ ( 589, 7, dummy_code = 0, "link seems to have been created");
590 call hcs_$delentry_file (parent, "x", 0); /* delete the link if it was placed */
591
592 /* Try to add a name to the directory */
593
594 reference = dir_pathx || ", adding the name of ""x""";
595 call hcs_$chname_file (parent, dirname, "", "x", code);
596 call hcs_$status_minf (parent, "x", 0, 0, 0, dummy_code);
597 call check_status_ ( 597, 8, dummy_code = 0, "name seems to have been added");
598 call hcs_$chname_file (parent, dirname, "x", "", code); /* this just cleans up the above */
599
600 /* Set the directory's ring brackets */
601
602 reference = dir_path;
603 call hcs_$set_dir_ring_brackets (parent, dirname, 6, code);
604 call hcs_$get_dir_ring_brackets (parent, dirname, drb, dummy_code);
605 call check_status_ ( 605, 51, dummy_code = 0 & (drb(1) ^= 7 | drb(2) ^= 7), "at least one of the ring brackets is no longer 7");
606 call hcs_$set_dir_ring_brackets (parent, dirname, 7, 0);
607
```

```
608 /* Delete the directory */
609
610 reference = dir_path;
611 call hcs_$delentry_file (parent, dirname, code);
612 call hcs_$status_mint (parent, dirname, 0, 0, 0, dummy_code); /* See if it was deleted */
613 /* Note: if the directory was in an upgraded parent higher than our authorization,
614     there is no way to tell if it really was deleted or not.  We can only check
615     if an illegal delete occurred if we had s permissin to do the status_mint above. */
616 call hcs_$append_branchx (parent, dirname, 01011b, 7, userid, 1, 0, 0, 0); /* put back directory in case it was deleted */
617 call check_status_ ( 617, 11, dummy_code ^= 0 & s_expected, "directory seems to have been deleted, as indicated by nonzero status
     code from hcs_$status_mint"); /* dummy_code from status_mint of zero indicates directory still there */
618
619 /* Delete the subtree in parent */
620
621 reference = parent;
622 call hcs_$del_dir_tree (parent_parent, parent_ename, code);
623 if ^upgrade then
624       if m_expected & code = error_table_$incorrect_access    /* this may occur if no access to parent_parent */
625       then code = 0;                                          /* which may be a legal case */
626 call hcs_$star_ (parent, "**", 3, areap, ecount, eptr, nptr, dummy_code); /* see if anything left in parent */
627 call hcs_$append_branchx (parent, dirname, 01011b, 7, userid, 1, 0, 0, 0); /* put back the directory in case deleted */
628 call hcs_$quota_move (parent_parent, parent_ename, saved_quota, 0); /* put back quota of parent */
629 /* Below, if we had s permission to use hcs_$star_, we can see if there was anything left
630     in parent by examining ecount.  If we didn't have s permission or couldn't
631     get star_ for some reason, the test for deletion can't be made. */
632 call check_status_ ( 632, 10, s_expected & ecount = 0 & dummy_code = 0, "subtree seems to have been deleted");
633 call reset_segment; /* put back the segment in case it was deleted */
634
635 /* Try to set the safety switch */
636
637 reference = dir_path;
638 call hcs_$set_safety_sw (parent, dirname, "1"b, code);
639 call hcs_$get_safety_sw (parent, dirname, safety_sw, dummy_code);
640 call check_status_ ( 640, 56, safety_sw ^= "0"b & dummy_code = 0, "safety switch was set");
641 call hcs_$set_safety_sw (parent, dirname, "0"b, code);
642
643 /* Try the quota move entry */
644
645 call hcs_$quota_move (parent, dirname, 1, code);  /* we expect no quota originally */
646 call hcs_$quota_get (dir_path, quota, trp, tup, infqcnt, taccsw, used, dummy_code);
647 call check_status_ ( 647, 43, s_expected & dummy_code = 0 & quota ^= 0,
648          "quota seems to have been moved.  Quota is ^d", quota);
649 call hcs_$quota_move (parent, dirname, -1, code); /* move quota back */
650 /*
```

```
*/
651 /* The following primitives require sma to the directory, but none to the parent.
652     They are only tested when upgrade is specified because the ACL of dirname is
653     always sma. */
654
655 if upgrade then do;
656       not_allowed_code = error_table_$incorrect_access;
657
658       /* Set the bitcount of the directory */
659
660       reference = dir_path;
661       call hcs_$set_bc (parent, dirname, 1, code);
662       call hcs_$status_mint (parent, dirname, 0, type, bc, dummy_code);
663       call check_status_ ( 663, 49, bc ^= 0 & dummy_code = 0 & s_expected, "bitcount was changed");
664       call hcs_$set_bc (parent, dirname, 0, code);  /* restore the bitcount */
665       end;
666 /*
```

```
*/
667 /* Now test the hcs_ calls that were intended to reference a segment, referencing
668     dir_path instead.  These tests are done
669     to insure that the error code returns no information about the status of the entry
670     if it shouldn't */
671
672 not_allowed_code = error_table_$incorrect_access;
673 allowed_code = error_table_$dirseg; /* if access was allowed, this error should occur */
674 reference = dir_path;
675 call check_status_$set (s, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
676
677 /* Try the list_acl for a segment */
678
679 call list_acl_test ( 679, 36, hcs_$list_acl, parent, dirname);
680
681 /* Try get_ring_brackets for a segment */
682
683 rb = -1;
684 call hcs_$get_ring_brackets (parent, dirname, rb, code);
685 call check_status_ ( 685, 30, sum(rb) ^= -3, "ring brackets (^a,^a,^a) were returned", rb(1),
686      rb(2), rb(3));
687
688 /* Try other acl commands for segments */
689
690 if m_expected
691 then allowed_code = error_table_$bad_ring_brackets;
692 else allowed_code = error_table_$incorrect_access;
693 call set_acl_test ( 693, 1, parent, dirname);      /* add_acl_entries */
694 call set_acl_test ( 694, 5, parent, dirname);      /* replace_acl */
695 call set_acl_test ( 695, 9, parent, dirname);      /* delete_acl_entries */
696
697 /* Try the append branch entries */
698
699 allowed_code = error_table_$namedup;
700 if ^upgrade then not_allowed_code = error_table_$namedup;
701 call hcs_$append_branch (parent, dirname, 0, code);
702 call check_status_ ( 702, 5, ""b, "");
703 call hcs_$append_branchx (parent, dirname, 0, (ring), "xxxxx.xxxxx.x", 0, 0, 0, code);
704 call check_status_ ( 704, 6, ""b, "");
705 if ai then do;
706      call hcs_$create_branch_ (parent, dirname, addr(branch_), code);
707      call check_status_ ( 707, 70, ""b, "");
708      end;
709
710 /* Try the move entry */
711
712 reference = dir_pathx || " as source";
713 allowed_code = error_table_$dirseg;
714 if ^upgrade then not_allowed_code = error_table_$moderr;
715 call hcs_$fs_move_file (parent, dirname, 0, pd, tempname, code);
716 call check_status_ ( 716, 21, ""b, "");
717
718 reference = dir_pathx || " as destination";
719 call hcs_$fs_move_file (pd, tempname, 0, parent, dirname, code);
720 call check_status_ ( 720, 21, ""b, "");
721
722 /* Try the initiate calls */
723
```

```
724 reference = dir_path;
725 segptr = null;
726 call hcs_$initiate (parent, dirname, "", 0, 0, segptr, code);
727 call check_status_ ( 727, 33, segptr ^= null, "pointer should not have been returned for a directory: ^p", segptr);
728 segptr = null;
729 bc = -1;
730 call hcs_$initiate_count (parent, dirname, "", bc, 0, segptr, code);
731 call check_status_ ( 731, 34, segptr ^= null | bc ^= 0, "information should not have been returned for a directory!
732 pointer ^a, bitcount ^a", conv(segptr), conv((bc)));
733
734 /*  To try the make_ptr entry, we set the search rules to include the parent
735     directory, and then call hcs_$make_ptr.  If we couldn't set the search rules,
736     we can't try anything. */
737
738 allowed_code = error_table_$dirseg;
739 if ^upgrade then not_allowed_code = allowed_code;
740 search_rules = saved_search_rules;
741 search_rules.num = search_rules.num + 1;
742 search_rules.names(search_rules.num) = parent;
743 segptr = null;
744 call hcs_$initiate_search_rules (addr(search_rules), code);
745 if code = 0 then do;
746     call hcs_$make_ptr (null, dirname, dirname, segptr, code);
747     call check_status_ ( 747, 40, segptr ^= null, "pointer to entry should not have been returned for directory! ^p", segptr);
748     end;
749 call hcs_$initiate_search_rules (addr(saved_search_rules), code); /* set them back */
750
751 /* Try to set the max length & ring brackets */
752
753 if m_expected
754 then allowed_code = error_table_$dirseg;
755 else allowed_code = error_table_$incorrect_access;
756 if ^upgrade then not_allowed_code = error_table_$incorrect_access;
757 call hcs_$set_max_length (parent, dirname, 0, code);
758 call check_status_ ( 758, 52, ""b, "");
759
760 call hcs_$set_ring_brackets (parent, dirname, 4, code);
761 call check_status_ ( 761, 54, ""b, "");
762
763 /* Try the terminate entries */
764
765 if upgrade then do; /* terminate is allowed if not upgrade */
766     call hcs_$terminate_file (parent, dirname, 0, code); /* this entry returns no error code */
767     call check_status_ ( 767, 63, ""b, "");
768     end;
769
770 allowed_code = error_table_$name_not_found;
771 not_allowed_code = error_table_$name_not_found;
772 call hcs_$terminate_name (dirname, code);
773 call check_status_ ( 773, 64, ""b, "");
774
775 /* Finally, try to truncate */
776
777 allowed_code = error_table_$dirseg;
778 if upgrade
779 then not_allowed_code = error_table_$incorrect_access;
780 else not_allowed_code = error_table_$dirseg;
781 call hcs_$truncate_file (parent, dirname, 0, code);
```

```
call check_status_ ( 782, 67, ""b, "");
/*
```

```
*/
784 /* This group of tests attempts to reference a segment in the parent
785    directory.  First we will reference the segment's branch with primitives that do
786    not require the segment to be initiated, and which require only s permission
787    to the parent directory.
788 */
789
790 reference = seg_path;
791 allowed_code = 0;              /* We expect no error if there is s permission */
792 if upgrade
793 then not_allowed_code = error_table_$incorrect_access;
794 else not_allowed_code = error_table_$incorrect_access;
795 call check_status_$set (s, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
796
797 /*  Try the various "get" entries */
798
799 scratch = "";
800 call hcs_$get_author (parent, segname, 0, scratch, code);
801 call check_status_ ( 801, 25, scratch ^= "", "author's name returned: ^a", scratch);
802 scratch = "";
803 call hcs_$get_bc_author (parent, segname, scratch, code);
804 call check_status_ ( 804, 26, scratch ^= "", "author's name returned: ^a", scratch);
805 rb = -1;
806 call hcs_$get_ring_brackets (parent, segname, rb, code);
807 call check_status_ ( 807, 30, sum(rb) ^= -3, "ring brackets (^a,^a,^a) were returned", rb(1),
808      rb(2), rb(3));
809 max_length = -1;
810
811 /* Try to use make_ptr by first setting the search rules for the parent */
812
813 search_rules = saved_search_rules;
814 search_rules.num = search_rules.num + 1;
815 search_rules.names(search_rules.num) = parent;
816 segptr = null;
817 allowed_code = error_table_$no_linkage; /* It's not an object segment, but access is allowed */
818 if ^upgrade then not_allowed_code = error_table_$no_linkage;
819 call hcs_$initiate_search_rules (addr(search_rules), code);
820 if code = 0 then do;   /* we can only make this test if the search rules can be set */
821    call hcs_$make_ptr (null, segname, segname, segptr, code);
822    call check_status_ ( 822, 40, segptr ^= null, "pointer to segment should not have been returned: ^p", segptr);
823    end;
824 call hcs_$initiate_search_rules (addr(saved_search_rules), code);
825
826 /* List the segment's ACL */
827
828 allowed_code = 0;
829 if ^upgrade then not_allowed_code = error_table_$incorrect_access;
830 call list_acl_test ( 830, 36, hcs_$list_acl, parent, segname);
831
832 /* Get the status of segment */
833
834 if ^upgrade then not_allowed_code = error_table_$no_s_permission;
835 branch = ""b;
836 call hcs_$status_ (parent, segname, 0, addr(branch), areap, code);
837 if upgrade then result = branch ^= ""b;
838           else result = (bbranch.pad1 | bbranch.pad2) ^= ""b;
839 call check_status_ ( 839, 59, result, "information about the branch was returned");
840
```

```
841 branch = ""b;
842 call hcs_$status_long (parent, segname, 0, addr(branch), areap, code);
843 if upgrade then result = branch ^= ""b;
844            else result = (bbranch.pad1 | bbranch.pad2 | bbranch.pad3 | bbranch.pad4) ^= ""b;
845 call check_status_ ( 845, 60, result, "information about the branch was returned");
846
847 /* There's no point in trying calls  that reference a segment that require
848    s permission and a pointer to the segment, since in order to get the
849    pointer we must have had s permission already.  This also is why it
850    makes no sense to test the terminate entries.
851 */
852 /*
```

```
*/
853 /* The following primitives require access to the segment, but not to the
854    parent.  They are only tested if upgrade is specified, because we
855    know we have rew to the segment.  The status code can reveal whether the segment exists. */
856
857 if upgrade then do;
858      reference = seg_path;
859      not_allowed_code = error_table_$incorrect_access;
860
861      /* Try the "get" entries */
862
863      call hcs_$get_max_length (parent, segname, max_length, code);
864      call check_status_ ( 864, 28, max_length ^= -1, "max length returned! ^d", max_length);
865
866      safety_sw = "1"b;
867      call hcs_$get_safety_sw (parent, segname, safety_sw, code);
868      call check_status_ ( 868, 31, safety_sw ^= "1"b, "safety switch was returned");
869
870      /* Try the initiate entries */
871
872      segptr = null;
873      call hcs_$initiate (parent, segname, "", 0, 0, segptr, code);
874      if code = error_table_$segknown then code = 0;
875      call check_status_ ( 875, 33, segptr ^= null, "pointer should not have been returned! ^p", segptr);
876      segptr = null;
877      bc = -1;
878      call hcs_$initiate_count (parent, segname, "", bc, 0, segptr, code);
879      if code = error_table_$segknown then code = 0;
880      call check_status_ ( 880, 34, segptr ^= null | bc ^= 0, "information should not have been returned:
881      pointer ^a, bitcount ^a", conv(segptr), conv((bc)));
882
883      /* Get the status that is allowed without s permission */
884
885      type, bc = -1;
886      call hcs_$status_minf (parent, segname, 0, type, bc, code);
887      call check_status_ ( 887, 61, type+bc ^= -2, "type or bitcount was returned! type = ^a, bitcount = ^a",
888           conv((type)), conv((bc)));
889
890      /* Move the segment */
891
892      reference = seg_pathx || " as source";
893      call hcs_$truncate_seg (temptr, 0, code); /* destination must be zero length */
894      call hcs_$fs_move_file (parent, segname, 10b, pd, tempname, code);
895      call get_segment;
896      if segptr ^= null then
897         call check_status_ ( 897, 21, temptr -> first_bit = "1"b | segptr -> first_bit = "0"b,
898                 "data was moved from segment or segment was truncated");
899      else call check_status_ ( 899, 21, temptr -> first_bit = "1"b, "data was moved from segment");
900      call reset_segment;
901      call hcs_$set_safety_sw (parent, dirname, "0"b, code);
902
903      temptr -> first_bit = "0"b;
904      call hcs_$truncate_seg (segptr, 0, code); /* this may not work if no access, but the move won't work either */
905      reference = seg_pathx || " as destination";
906      call hcs_$fs_move_file (pd, tempname, 0, parent, segname, code);
907      call get_segment;
908      if segptr ^= null then
909         call check_status_ ( 909, 21, segptr -> first_bit = "0"b, "data was moved into segment");
```

```
910      else call check_status_ ( 910, 21, ""b, "");
911      call reset_segment;
912
913      /* Try to set the bitcount */
914
915      reference = seg_path;
916      call hcs_$set_bc (parent, segname, 0, code);
917      call get_segment;
918      call check_status_ ( 918, 49, s_expected & dummy_code = 0 & bc ^= 1, "bitcount was changed");
919      call hcs_$set_bc (parent, segname, 1, code);
920
921      /* The last thing to test is the truncate entry */
922
923      call hcs_$truncate_file (parent, segname, 0, code);
924      call get_segment;
925      if segptr ^= null then
926         call check_status_ ( 926, 67, segptr -> first_bit ^= "1"b, "segment seems to have been truncated");
927      else call check_status_ ( 927, 67, ""b, "");
928      call reset_segment;
929      end;
930 /*
```

```
*/
931 /* Try the segment references requiring m permission to the parent.  Some of these
932     require a pointer to the segment, which we may not have if we didn't have s permission.
933     Therefore, first try the ones that don't require a pointer */
934
935 if upgrade
936 then not_allowed_code = error_table_$incorrect_access;
937 else not_allowed_code = error_table_$incorrect_access;          /* if we didn't have s or m */
938 reference = seg_path;
939 allowed_code = 0;
940 call check_status_$set (m, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
941
942 /* Try the ACL entries that modify the ACL */
943
944 call set_acl_test ( 944, 1, parent, segname);     /* add_acl_entries */
945 call set_acl_test ( 945, 5, parent, segname);     /* replace_acl */
946 call set_acl_test ( 946, 9, parent, segname);     /* delete_acl_entries */
947
948 /* Append a branch.  Create the dummy segment with the name of "x". */
949
950 reference = x_path;
951 call hcs_$append_branch (parent, "x", 0, code);
952 call check_status_ ( 952, 5, ""b, "");
953 call hcs_$delentry_file (parent, "x", code); /* delete it if it was created */
954 call hcs_$append_branchx (parent, "x", 0, (ring), "xxxxx.xxxxx.x", 0, 0, 0, code);
955 call check_status_ ( 955, 6, ""b, "");
956 call hcs_$delentry_file (parent, "x", code); /* delete again */
957
958 /* Add a name to segment */
959
960 reference = seg_pathx || " changing the name to ""x""";
961 call hcs_$chname_file (parent, segname, "", "x", code);
962 call check_status_ ( 962, 8, ""b, "");
963 call hcs_$chname_file (parent, segname, "x", "", code); /* remove the name */
964
965 /* Delete the segment */
966
967 reference = seg_path;
968 call hcs_$delentry_file (parent, segname, code);
969 call check_status_ ( 969, 11, ""b, "");
970 call reset_segment;
971
972 /* Use make_seg to create a segment */
973
974 dcl xptr ptr;
975 xptr = null;
976 reference = x_path;
977 call hcs_$make_seg (parent, "x", "", 01110b, xptr, code);
978 call check_status_ ( 978, 41, xptr ^= null, "pointer to segment was returned: ^p", xptr);
979 call hcs_$delentry_file (parent, "x", code); /* delete it once more */
980
981 /* Try the "set" entries */
982
983 reference = seg_path;
984 call hcs_$set_max_length (parent, segname, 2048, code);
985 call hcs_$get_max_length (parent, segname, max_length, dummy_code);
986 call check_status_ ( 986, 52, s_expected & dummy_code = 0 & max_length ^= 1024, "max length was set");
987 call hcs_$set_max_length (parent, segname, 1024, code);
```

```
988
989 call hcs_$set_ring_brackets (parent, segname, 5, code);
990 call hcs_$get_ring_brackets (parent, segname, rb, dummy_code);
991 call check_status_ ( 991, 54, dummy_code = 0 & s_expected & (rb(1)^=4 | rb(2)^=4 | rb(3)^=4), "ring brackets were changed");
992 call hcs_$set_ring_brackets (parent, segname, 4, dummy_code);
993
994 call hcs_$set_safety_sw (parent, segname, "1"b, code);
995 call hcs_$get_safety_sw (parent, segname, safety_sw, dummy_code);
996 call check_status_ ( 996, 56, dummy_code = 0 & s_expected & safety_sw, "safety switch was set");
997 call hcs_$set_safety_sw (parent, segname, "0"b, code);
998 /*
```

```
*/
 999 /* Now we can test those segment references that require m permission, and
1000     which require the segment to first be initiated
1001 */
1002
1003 /* First, see if we can initiate */
1004
1005 call get_segment;
1006 if segptr = null then goto last_set; /* if we can't initiate, don't try these tests */
1007
1008 /* Try all the "_seg" calls that need m permission */
1009
1010 call hcs_$chname_seg (segptr, "", "x", code);
1011 call check_status_ (1011, 9, ""b, "");
1012 call hcs_$chname_seg (segptr, "x", "", code);
1013
1014 call hcs_$delentry_seg (segptr, code);
1015 call check_status_ (1015, 12, ""b, "");
1016 cal. reset_segment;
1017
1018 /* There's no point in trying hcs_$fs_move_seg or hcs_$set_bc_seg
1019     because they are legal without access to the directory.  The only time
1020     they are illegal is when upgraded, but in that case we could have never
1021     gotten a pointer in the first place. */
1022 /*
```

```
*/
1023 /* This last series of tests make reference to the branch of the segment,
1024    but are intended for directory references */
1025
1026 last_set:
1027
1028 reference = seg_path;
1029 allowed_code = error_table_$notadir;
1030 if upgrade
1031 then not_allowed_code = error_table_$incorrect_access;
1032 else not_allowed_code = error_table_$notadir;
1033 call check_status_$set (s, mode_expected, addr(code), addr(allowed_code), addr(not_allowed_code), end_all, addr(reference));
1034
1035 call set_acl_test (1035, 4, parent, segname);                /* add_dir_inacl_entries */
1036 call set_acl_test (1036, 3, parent, segname);                /* add_inacl_entries */
1037 call set_acl_test (1037, 12, parent, segname);     /* delete_dir_inacl */
1038 call set_acl_test (1038, 11, parent, segname);     /* delete_inacl */
1039 call set_acl_test (1039, 8, parent, segname);                /* replace_dir_inacl */
1040 call set_acl_test (1040, 7, parent, segname);                /* replace_inacl */
1041 if m_expected
1042 then allowed_code = error_table_$bad_ring_brackets;
1043 else allowed_code = error_table_$incorrect_access;
1044 if ^upgrade then not_allowed_code = error_table_$incorrect_access;
1045 call set_acl_test (1045, 2, parent, segname);                /* add_dir_acl_entries */
1046 call set_acl_test (1046, 10, parent, segname);     /* delete_dir_acl */
1047 call set_acl_test (1047, 6, parent, segname);                /* replace_dir_acl */
1048
1049 allowed_code = error_table_$nondirseg;
1050 call list_acl_test (1050, 37, hcs_$list_dir_acl, parent, segname);
1051 allowed_code = error_table_$notadir;
1052 if ^upgrade then not_allowed_code = allowed_code;
1053 call list_acl_tast (1053, 38, hcs_$list_dir_inacl, parent, segname);
1054 call list_acl_test (1054, 39, hcs_$list_inacl, parent, segname);
1055
1056 allowed_code = error_table_$namedup;
1057 if ^upgrade then not_allowed_code = allowed_code;
1058 call hcs_$append_branchx (parent, segname, 0, (ring), "xxxxx.xxxxx.x", 1, 0, 0, code);
1059 call check_status_ (1059, 6, ""b, "");
1060 if ai then do;
1061       call hcs_$create_branch_ (parent, segname, addr(branch_), code);
1062      call check_status_ (1062, 70, ""b, "");
1063      end;
1064
1065 allowed_code = error_table_$notadir;
1066 if ^upgrade then not_allowed_code = allowed_code;
1067 call hcs_$del_dir_tree (parent, segname, code);
1068 call check_status_ (1068, 10, ""b, "");
1069
1070 call hcs_$fs_search_set_wdir ((seg_path), code);
1071 call check_status_ (1071, 24, ""b, "");
1072 call hcs_$fs_search_set_wdir ((wdir), code);
1073
1074 if ^upgrade then not_allowed_code = error_table_$incorrect_access;
1075 call hcs_$get_dir_ring_brackets (parent, segname, drb, code);
1076 call check_status_ (1076, 27, ""b, "");
1077
1078 if ^upgrade then not_allowed_code = error_table_$notadir;
1079 call hcs_$quota_get (seg_path, quota, trp, tup, infqcnt, taccsw, used, code);
```

```
call check_status_ (1080, 35, ""b, "");

if m_expected
then allowed_code = error_table_$notadir;
else allowed_code = error_table_$incorrect_access;
if ^upgrade then not_allowed_code = error_table_$incorrect_access;
call hcs_$quota_move (parent, segname, -1, code);
call check_status_ (1087, 43, ""b, "");

call hcs_$set_dir_ring_brackets (parent, segname, 7, code);
call check_status_ (1090, 51, ""b, "");
/*



*/
/******* END OF PROGRAM ********/

call cleanup_stuff;
return;

/* Come here on any error */

end_all: error = -2;
         call cleanup_stuff;
         return;
/*
```

```
*/
/* This procedure is called to test the add, delete, and replace acl or inacl calls.
   The argument "type" specifies which call is to be made (see l(type) below).

   For the add entries, the structure defined in the main program called
   dir_acl or segment_acl is used.  This should contain a dummy user-project id.
   For the replace entries, the saved_acl (saved_dir_acl, saved_seg_acl, saved_seg_inacl,
   or saved_dir_inacl) structure, saved during initialization in the main program,
   is used so that the acl is not modified in case replacement was allowed.
   For the delete_acl, the same delete_acl structure defined in the main program
   is used for all calls.
*/
set_acl_test: proc (loc, type, dirname, ename);
dcl (dirname, ename) char(*);
dcl loc fixed bin;
dcl type fixed bin;
dcl numbers(12) fixed bin static init (1,2,4,3,44,45,47,46,13,14,16,15);

delete_acl(1).code = -1;

goto l(type);

l(1):     call hcs_$add_acl_entries (dirname, ename, addr(segment_acl), 1, code);
          goto common;
l(2):     call hcs_$add_dir_acl_entries (dirname, ename, addr(dir_acl), 1, code);
          goto common;
l(3):     call hcs_$add_inacl_entries (dirname, ename, addr(segment_acl), 1, ring, code);
          goto common;
l(4):     call hcs_$add_dir_inacl_entries (dirname, ename, addr(dir_acl), 1, ring, code);
          goto common;
l(5):     call hcs_$replace_acl (dirname, ename, addr(saved_seg_acl), saved_seg_acl_count, "0"b, code);
          goto common;
l(6):     call hcs_$replace_dir_acl (dirname, ename, addr(saved_dir_acl), saved_dir_acl_count, "0"b, code);
          goto common;
l(7):     call hcs_$replace_inacl (dirname, ename, addr(saved_seg_inacl), saved_seg_inacl_count, "0"b, ring, code);
          goto common;
l(8):     call hcs_$replace_dir_inacl (dirname, ename, addr(saved_dir_inacl), saved_dir_inacl_count,
               "0"b, ring, code);
          goto common;
l(9):     call hcs_$delete_acl_entries (dirname, ename, addr(delete_acl), 1, code);
          goto common;
l(10):    call hcs_$delete_dir_acl_entries (dirname, ename, addr(delete_acl), 1, code);
          goto common;
l(11):    call hcs_$delete_inacl_entries (dirname, ename, addr(delete_acl), 1, ring, code);
          goto common;
l(12):    call hcs_$delete_dir_inacl_entries (dirname, ename, addr(delete_acl), 1, ring, code);

common:                          /* all of above calls return here */
if type < 9 then
   call check_status_ /* space */ (loc, numbers(type), ""b, "");
else do; /* if s permission was not expected, no information should be returned about the ACL names */
   call check_status_ /* space */ (loc, numbers (type), delete_acl(1).code ^= 0 & ^s_expected,
                    "code ""^a"" set in delete_acl structure for user ^a",
                    convert_status_code_ (delete_acl(1).code, "xxxxxxxx"), delete_acl(1).name);
   end;
end;
/*
```

```
*/
/* This subroutine performs the list_acl tests.  It is called with the entry number,
   the entry value, and the pathname whose acl is to be referenced.
   It takes care of storage managment and error messages that may be required. */

list_acl_test: proc (loc, n, entry, dirname, ename);
 dcl loc fixed bin; /* line number where we came from */
 dcl n fixed bin;              /* entry number */
 dcl entry entry;              /* entry value */
 dcl entry_inacl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin(3), fixed bin(35)) based (addr(entry));
 dcl entry_acl entry (char(*), char(*), ptr, ptr, ptr, fixed bin, fixed bin(35)) based (addr(entry));
 dcl (dirname, ename) char(*);
 dcl 1 acl(acl_count) based (area_ret_ptr),
       2 name char(32);
 dcl area_ret_ptr ptr init (null);
 dcl acl_count fixed bin init(-1);

 on cleanup begin;
      if area_ret_ptr ^= null then free acl;
      end;

 if entry = hcs_$list_inacl | entry = hcs_$list_dir_inacl then
      call entry_inacl (dirname, ename, areap, area_ret_ptr, null, acl_count, (ring), code);
 else
      call entry_acl (dirname, ename, areap, area_ret_ptr, null, acl_count, code);
 if area_ret_ptr ^= null & acl_count <= 0 then
      call check_status_ /* space */ (loc, n, "1"b, "pointer to ACL structure was returned, but no ACL info");
 if area_ret_ptr ^= null & acl_count > 0 then
      call check_status_ /* space */ (loc, n, "1"b, "pointer to ACL structure was returned, and count of ^d was returned.
First name on ACL: ""^a""", acl_count, acl(1).name);
 if area_ret_ptr = null then call check_status_ /* space */ (loc, n, ""b, "");
 if area_ret_ptr ^= null then free acl;
end;
/*
```

```
*/
/* This subroutine is called after any operation that modifies (or might have
   modified) the segment at seg_path.  It creates the segment, if it doesn't still
   exist, and sets the first bit of the segment to "1"b.
*/
reset_segment: proc;
  dcl (no_write_permission, not_in_write_bracket) condition;
  dcl code fixed bin(35);

call hcs_$make_seg (parent, segname, "", 01110b, segptr, code);
if segptr = null then return; /* We had no access in the first place--segment must not have been modified */

on no_write_permission goto ignore;
on not_in_write_bracket goto ignore;

segptr -> first_bit = "1"b;  /* try to restore the first bit.  If we didn't have write permission,
                                we must not have changed it. */
ignore:
  call hcs_$set_bc_seg (segptr, 0, code); /* bitcount must be zero */
  call hcs_$set_ring_brackets (parent, segname, 4, code); /* rings of 4 4 4 */
  call hcs_$set_safety_sw_seg (segptr, "0"b, code); /* no safety switch */

end;


/* Procedure to return the bitcount and a pointer, and a code for the segment */

get_segment: proc;
  call hcs_$initiate_count (parent, segname, "", bc, 0, segptr, dummy_code);
end;
/*
```

```
*/
/* Cleanup procedure to restore anything that might have been changed and
   free storage */

cleanup_stuff: proc;

/* Delete the name "x", if it appears anywhere.  Also delete any entries named "x" */

call hcs_$chname_file (parent, "x", "x", "", code);
call hcs_$delentry_file (parent, "x", code); /* delete a branch that might have been placed */
call reset_segment;

/* restore all ACLs */

if saved_search_rules.num ^= 0 then
     call hcs_$initiate_search_rules (addr(saved_search_rules), code);
if saved_seg_acl_ptr ^= null & saved_seg_acl_ptr ^= addr(segment_acl) then do;
     call hcs_$replace_acl (parent, segname, saved_seg_acl_ptr, saved_seg_acl_count, "1"b, code);
     free saved_seg_acl;
     end;
if saved_dir_acl_ptr ^= null & saved_dir_acl_ptr ^= addr(dir_acl) then do;
     call hcs_$replace_dir_acl (parent, dirname, saved_dir_acl_ptr, saved_dir_acl_count, "1"b, code);
     free saved_dir_acl;
     end;
if saved_seg_inacl_ptr ^= null & saved_seg_inacl_ptr ^= addr(segment_acl) then do;
     call hcs_$replace_inacl (parent, dirname, saved_seg_inacl_ptr, saved_seg_inacl_count, "1"b, ring, code);
     free saved_seg_inacl;
     end;
if saved_dir_inacl_ptr ^= null & saved_dir_inacl_ptr ^= addr(dir_acl) then do;
     call hcs_$replace_dir_inacl (parent, dirname, saved_dir_inacl_ptr, saved_dir_inacl_count, "1"b, ring, code);
     free saved_dir_inacl;
     end;

/* make quota on dirname zero */

call hcs_$quota_get (dir_path, quota, 0, ""b, 0, 0, 0, code);
call hcs_$quota_move (parent, dirname, -quota, code);

/* restore quota on parent in case it was changed */

if saved_quota ^= 0 then do;
   call hcs_$quota_get (parent, quota, 0, ""b, 0, 0, 0, code);
   call hcs_$quota_move (parent_parent, parent_ename, saved_quota - quota, code);
   end;

call hcs_$delentry_seg (temptr, code);
call change_wdir_ (wdir, code);
if eptr ^= null then free entries;
if nptr ^= null then free names;
call reset_segment;
end;




end;
```

REFERENCES

1. "ESD 1974 Computer Security Developments Summary", MCI-75-1, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, December, 1974.

2. W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45", ESD-TR-75-69, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, May, 1975.

3. D.E. Bell and E.L. Burke, "A Software Validation Technique for Certification, Part 1: The Methodology", ESD-TR-75-54, Volume I, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, April, 1975.

4. P.A. Karger and R.R. Schell, "Multics Security Evaluation: Vulnerability Analysis", ESD-TR-74-193, Volume II, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, June, 1974.

5. Honeywell Information Systems, "Design For Multics Security Enhancements", ESD-TR-74-176, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, 1974.

6. D.E. Bell and L.J. LaPadula, "Secure Computer Systems", ESD-TR-73-278, Volume I-III, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, November, 1973 - June 1974.

7. M. Gasser, "A Random Word Generator", ESD-TR-75-97, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, November 1975.

8. D.L. James, "A Remote Terminal Emulator for Loading and Performance Measurement of On-Line Systems", ESD-TR-75-97, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, November 1975.

9. K. Hennigan, "Hardware Subverter for the Honeywell 6180", ESD-TR-76-352, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Massachusetts, December 1976.

10. Multics Programmers' Manual, AG91, AG92, AG93 and AK92, Honeywell Information Systems Inc., 1975.